Formal Methods in HCI: a Success Story – why it works and how to reproduce it

Alan Dix

Computing Department Lancaster University
Lancaster, LA1 4YR, UK

(unpublished manuscript Jan. 2002)

http://www.hcibook.com/alan/papers/formal-2002/

Abstract

Strong success stories of formal methods are relatively rare. This paper discusses a case study where the use of formal dialogue specification improved the efficiency of production by an order of magnitude. The actual example comes from some years ago, but this paper seeks to understand the reasons for success so that they can be reapplied to more advanced, but still often under-utilised, methods today. Two issues are dealt with in detail: interface state and the blending of formal and informal representations. The issues of state are also discussed in relation to the particular problems of web interfaces.

Keywords: formal methods, dialogue specification, web interfaces

1. Introduction

There is a strong community working on the use of formal methods within HCI (see for example Palanque and Paterno's collection [1997] or the DSVIS conference series). However, there is still a strong perception, both within HCI and in computing in general that formal methods are (i) difficult and expensive to apply and (ii) only really useful in safety critical domains such as air traffic control. In fact, there are a significant number of commercial success stories in formal methods in general (Clarke. 1996) and considerable interest within commercial user-interface design, although it must be said, mainly in the safety critical domain.

In this paper we're going to look at a case study of successful use of formal methods in HCI in a non-safety critical area – standard data processing. However, we'll not be looking at a recent case study using the most up-to-date methods, but instead one from over 15 years ago, in fact before I became a computing academic, and before I'd even heard the term HCI! At the time I was working for Cumbria County Council working on transaction processing programs in COBOL, the sort of thing used for stock control or point-of-sale terminals in large organisations.

Why such an old example, rather than a more sexy and up-to-date one? Well first because this sort of system is still very common. In addition, the

issues in these large centralised transaction processing systems are very similar to those of web-based interfaces, especially e-commerce systems. Thirdly, it is a resounding success story, which is not too common in this area, and a 1000% performance improvement is worth shouting about. Finally and most significantly, because it was such a success, it gives us a chance to analyse why it was so successful and what this tells us about using formalism today.

The other thing I ought to note is that although this was a very successful application of formal methods in interface design, I didn't understand *why* at the time. It is only comparatively recently that I've come to understand the rich interplay of factors that made it work and so perhaps be able to produce guidelines to help reproduce that success. So, that is why this paper is being written today!

Most of my own work in formal methods has been on suing them to produce general models and obtain understanding of broad issues of HCI. The success criteria for this work is very different, measured in understanding and insight. However, in this paper the focus is very much on the use of formal specification within the design of specific system and success criteria is in terms of time saved and pound notes!

We'll first of all look at the case study problem and the formal methods solution to it. This leads to a large number of success factors. Two strands will be examined in details – the issue of interface state, in section 4, and blending formal and informal representation, in section 5. In section 4 we will also look at the issue of state within web interface design.

2. Case study – the problem…

2.1 transaction processing

Transaction processing systems such as IBM CICS have been the workhorses of large-scale information systems, stock management and order processing since the late 1970's. They are designed to be able to accept many thousands of active users.

Architecturally these systems are based around a central server (or cluster) connected to corporate databases and running the transaction-processing engine. In the system I worked with this was an ICL mainframe but in web-base applications will simply be a web server or enterprise server. The user interacts with a form-based front-end. In the systems I dealt with in the mid-80s the front-end was semi-intelligent terminals capable of tabbing between fields. Subsequently, in many areas these were replaced by PCs running 'thin client' software and now may be web-based forms. The centralisation of data and transaction processing ensures integrity of the corporate data, but the fact that users interact primarily with local terminals/PCs/browsers means that the central server does not have to manage the full load of the users interactions.

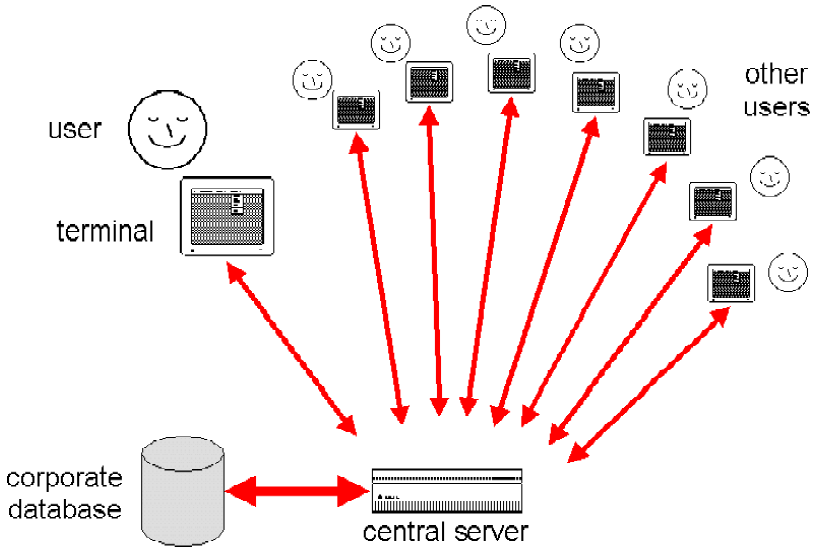


Figure 1. physical architecture of transaction processing system

When the user interacts the succession of events is as follows:

① user fills in form on terminal
② terminal may perform some initial validation (e.g. number vs. letters, range checks, date format, or, on thin PC client or Javascript on web form, more complex validation)
③ user checks and then submits form (presses special key or screen button)
④ terminal/PC/browser sends form data as a message to the transaction processing engine (e.g. CICS or web server) on the central server
⑤ transaction-processing engine selects appropriate application module for message (based on last screen/web page or information in message)
⑥ application module interprets message (form data), does further checks, performs any database updates, gets any required data from the database and generates a new screen/web page as 'result'
⑦ transaction processing engine passes this back tot he terminal
⑧ terminal presents the screen/web page to the user

All these stages except ⑥ are managed by the transaction-processing infrastructure. This sounds as if the job in designing this part should be straightforward, most of the complexity of dealing with detailed user interactions have been dealt with. But it is not quite so simple as all that ...

2.2 the problem…

In a GUI or any single user interface, the succession of events in the program is straightforward:

- user event 1 arrives (e.g. mouse press)
- deal with event and update display
- user event 2 arrives (e.g. mouse release)
- deal with event and update display

- user event 3 arrives (e.g. key click)
- deal with event and update display

As we know this can cause enough problems!

In a transaction processing system, with one user, the application module may receive messages (with form data) in a similar fashion. However, the whole point of such systems is that they have many users. So, the module may receive messages from different users interleaved:

- message 1 for user A received
- deal with message and generate new screen/web page for user A
- message 1 for user B received
- deal with message and generate new screen/web page for user B
- message 2 for user B received
- deal with message and generate new screen/web page for user B
- message 2 for user A received
- deal with message and generate new screen/web page for user A

The transaction processing engine deals with passing the new screens back to the right user, but the application module has to do the right things with the form data in the messages. In the case of simple transactions, this may not be a problem, for example, if the application simply allows the user to enter an ISBN number and then returns data about the book, the system can simply deal with each message in isolation. However, a more complex dialogue will require some form of state to be preserved between transactions. For example, a request to delete a book may involve an initial screen where the user fills in the ISBN, followed by a confirmation screen showing the details of the book to be deleted. Only then, if confirmed, will the system actually do the deletion and generate a 'book has been deleted' screen. Even a search request that delivers several pages of results needs to keep track of which result page is required and the original search criteria.

Getting back to Cumbria in the mid-80s, the transaction systems in place at that stage only dealt with the simple stateless record display transactions or multi-page search transactions ... and even the latter had problems. When several users tried to search the same database using the system they were likely to get their results mixed up with one another!

I was charged with producing the first update system. Whilst occasionally getting someone else's search results was just annoying, deleting the wrong record would be disastrous.

2.3 all about state

So what was wrong with the existing systems and how could I avoid similar, but more serious problems? In essence, it is all about state.

In most computer programs you don't need to worry too much about state. You put data in a variable at one stage and at a later point if you require the data it is still there in the variable. However, in the case of transaction processing modules the module may be re-initialised between each transaction (as is the case with certain types of web CGI script), so values put in a variable during one transaction won't be there at all for the next transaction. Even worse, if the same module is dealing with several users values left behind from a transaction for one users may still be 'lying around' when the next user's transaction is processed. This is precisely what was happening in the

search result listings. Some of the state of the transaction (part of the search criteria) was being left in a variable. When the system was tested (with one user!), there was no problem, but when several users used the system their search criteria got mixed up. Although it was possible to explicitly save and restore data associated with a particular terminal/user, the programmers had failed to understand what needed to be stored. Instead, the exiting programs coped by putting state information into fields on the form that were then sent back to the next stage of the transaction. With web-based interfaces similar problems occur with session state.

However, there is also a second, more subtle part of the state the current location in the human–computer dialogue.

In traditional computer algorithmics, the location in the program is implicit. It is only when one starts to deal with event-driven systems, such as GUIs, network applications and transaction processing, that one has to explicitly deal with this. And of course traditional computer science training does little to help. Not only are the principle algorithms and teaching languages sequential, but also the historical development of the subject means that sequential structures such as loops, recursion, etc. are regarded as critical and in lists of essential competency whereas event-driven issues are typically missing. Even worse, event-based languages such as Visual Basic and other GUI development languages have been regarded as 'dirty' and not worthy of serious attention. Possibly this is changing with Java becoming a major teaching language, but still the event-driven features are low on the computer science agenda!

So, computer programmers in the mid-80s as well as today are ill prepared both conceptually and in terms of practical skills to deal explicitly with state, especially flow of control.

This was evident in the buggy transaction modules I was dealing with. The flow of the program code of each module looked rather like a twiggy tree, with numerous branches and tests that were effectively trying to work out where in the flow of the user interaction the transaction was situated.

```
if  confirm_field is empty  // can't be confirm screen
                  // or user didn't fill in the Y/N
box
then if  record_id is empty  // must be initial entry
     then prepare 'which record to delete' screen
     else if valid record_id
          then read record and prepare confirm screen
          else prepare error screen
else if  confirm_field = "Y'
     then if  record_id is empty  // help malformed
          then prepare error screen
          else  if valid record_id
                 else do deletion
                 then prepare error screen
     else if  confirm_field = "N'
          then prepare 'return to main menu' screen
          else prepare 'must answer Y/N' screen
```

No wonder there were bugs!

Of course, if one looks at many GUIs or web applications the code looks just the same ... Try using the back key or bookmarking an intermediate page in most multi-stage web forms and you'll probably find just how fragile the code is.

3. Case study – the solution

3.1 Flowcharts of dialogue

A flow chart of the program looked hideous and was very uninformative because the structure of the program was not related to the structure if the user interaction. So, instead of focusing on the code I focused on the user interaction and produced flowcharts of the human–computer dialogue. Figure 2 shows a typical flowchart.

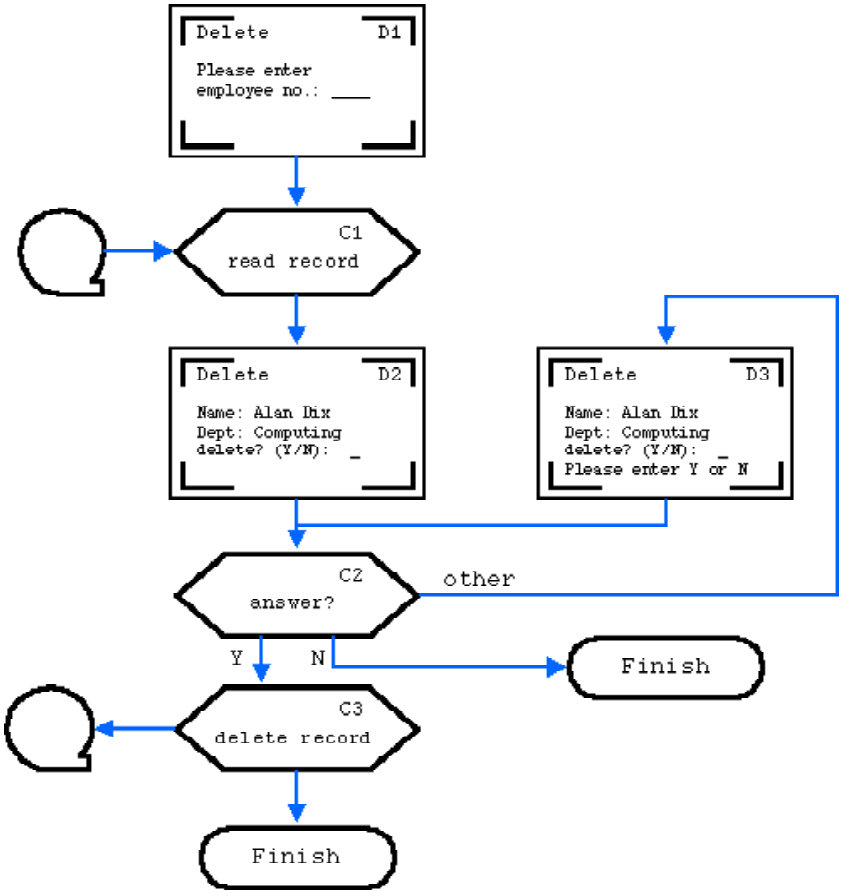


Figure 2. flow chart of user interaction

Each rectangle represents a possible screen and a miniature of the screen is drawn. The angled boxes represent system actions and the 'tape' symbols

represent database transactions. Note that this is not a flowchart of the program, but of the human–computer dialogue – it is a formal dialogue notation (although I didn't know the term at the time). Note also that the purpose is to clarify the nature of the dialogue, so the system side is only labeled in quite general terms (e.g. 'read record'). These labels are sufficient to clearly say what should happen and does not give all the details of how the code to do this works in details. This is because the difficult thing is getting the overall flow right.

Notice also that each major system block and each screen is labeled: D1, D2, D3 for the screens, C1, C2, C3 for the system code blocks. These are used to link the flowchart to boilerplate code. For each screen there is a corresponding block of code, which generates the screen and, very importantly, stores the label of the next system block against the terminal/user. For example, screen D3 will save the label 'C2'. The first thing the module does when asked to deal with a message is to retrieve the label associated with the user. If this is blank it is the start of the dialogue (generate screen D1 in this case), otherwise the module simply executes the code associated with the relevant system block.

This all seems very mundane, but the important thing is that it worked. Systems that were taking months to develop could be completed in days and the turnaround time for upgrades and maintenance was hours. That is systems were being produced at least **10 times faster** than previously and furthermore with less bugs!

3.2 Why it worked …

So why is it that such a simple formal method worked so well and can we use this to assess or improve other formalisms or develop new ones?

Let's look at some of the features that made it function well:

useful – addresses a real problem!

The notation focused on the overall user-interface dialogue structure that was causing difficulties in the existing systems. So often formalisms are proposed because they have some nice intrinsic properties, or are good for something else, but do not solve a real need.

appropriate – no more detailed than needed

For example, there was no problem in producing the detailed code to access databases etc., so the formalism deals with this at a very crude level 'read record', 'delete record' etc. Many formalisms force you to fill in lots of detail which makes it hard to see the things you really need it for as well as increasing the cost of using it.

communication – mini-pictures and clear flow easy to talk through with client

Formal methods are often claimed to be a means to improve communication within a design team, because of their precision. However, when precision is achieved at the cost of comprehensibility there is no real communication. Note also this was appropriate for communication with developers as it used a notation they were familiar with.

complementary – different paradigm than implementation

It is quite common to use specification methods that reflect closely the final structure of the system. For example, object-oriented specification for

object-oriented systems. Here however, the specification represents the structure of the dialogue which is completely different from the structure of the code. This is deliberate, the notation allows one to see the system from a different perspective. In this case one more suitable for producing and assessing the interface design. The relationship between the structure of the notation and the structure of the code is managed via simple rules, which is what formalisms are good at!

fast pay back – quicker to produce application (at least 1000%)

I have been involved in projects where substantial systems have been fully specified and then implemented and have seen the improvements in terms of quality and *long-term* time savings. However, I still rarely use these methods in practice even though I know they will save time. Why? because I, like most people, like instant pay-back. Spending lots of time up-front for savings later is very laudable, but when it comes to doing things I like to see results. Not only that, but clients are often happier to see a buggy partial something than to be told that ,yes, in a few months it will all come together. The dialogue flowcharts didn't just produce long-term savings, but also reduced the lead time to see the first running system.

responsive – rapid turnaround of changes

The feeling of control and comprehension made it easier to safely make changes. In some formal methods, the transformation process between specification and code is so complex that change is very costly (see [Dix 1989]). The assumption underlying this, as in much of software engineering, is that well specified systems will not need to be changed often. Of course, with user interfaces, however well specified, it is only when they are used that we really come to fully understand the requirements.

reliability – clear boiler plate code less error-prone

Although the transformation process from diagram to code was not automated, it was a fairly automatic hand process applying and modifying boiler plate code templates. This heavy reuse of standard code fragments greatly increases the reliability of code.

quality – easy to establish test cycle

The clear labeling of diagrams and code made it easy to be able to track whether all paths had been tested. However, note that these are not just paths through the program (which effectively restarted at each transaction), but each path through the human–computer dialogue.

maintenance – easy to relate bug/enhancement reports to specification and code

The screen's presented to the user included the labels making it easy to track bug reports or requests for changes both in the code and specification.

In short the formalism was used to fulfil a purpose, and was, above all, neither precious nor purist!!

4. More about state

4.1 Is state really that difficult?

When the bugs found in the system were described in section 2.3, did you think "obviously not very good developers that could make that sort of mistake"? In fact, these are not just common problems, but normal ones.

A simple example I use with students and in HCI tutorials is a four function calculator. What is in the state? Clearly there is a number that is currently displayed, but many students get stuck there. The parts that have no immediate display are harder think about as a designer. However, this hidden state is also more confusing for the user when it behaves unexpectedly, so it is more important that the designer gets it right.

Some of the extra detail becomes apparent when you think about particular user actions – when the '=' key is pressed the calculator must know what operation to do (+,-,*,/) and what the previous value was, so the state must record a 'pending' operation and a running total.

Another method, which I find very useful, is to play through a scenario annotated with the values of various state variables. Doing this to the calculator shows that an additional flag is needed to distinguish the case when the display says '2' because you just typed '2' or because you just typed '1+1'. In the former case typing '3' would give you '23', in the latter it would be '3'.

Calculator scenario

user types: 1 + 2 7 = - 3

start after 1 + 2

action	**display**	**pend_op**	**total**
	2	+	1
digit(7)	2 7	+	1
equals	2 8	none	2 8
op(-)	2 8	-	2 8
digit(3)	**283 !!!**	-	2 8

Figure 3 Calculator scenario

Let's just recap on this. I said that many students get stuck on at a single state variable for current display. Furthermore, in a group, even of computing masters students, typically **none** of them get all three of the state variables (display, running total and pending operation) without prompting from me to use one of the above methods. Remember too that most of these students will have had training in areas such as object oriented methods that should help in isolating state variables. And as for the 'am I in the middle of typing a number' flag – not only have I never found anyone who has produced this, when I originally started using this example I missed it and only noticed when I ran through the above scenario.

Recall that when we discussed the state of the transaction processing system in section 3.3 there were two kinds:

- temporary values such as the current location in a search result display
- the value representing the current location in a dialogue

The three 'easier' state variables were of the first kind. The flag, that even I forgot, was of the second.

So clearly state is very difficult to understand, and 'where we are' state is most difficult of all.

In section 2.3, I noted that this was partly because computing training does not focus on the sorts of systems where you need to think so explicitly; on state. When you want a variable you just name it and, most of the time, it has the right lifetime without worrying too much. (In fact, many Java programs are full of variables declared at an object level that should really be method variables, exactly the same problem as in the TP systems. It doesn't show up as a bug unless the object is used simultaneously by several threads.)

The second type of state, 'where we are', is represented explicitly in the computer itself in the form of the program counter. However, the programmer see the text of the program as a whole, the program counter is implicit.

In real life too we find that state which is apparent in the external attributes of an object (such as the size of a balloon, or the height of a ball) are easy to think about, whereas those that are hidden (such as the velocity of the ball) are much more difficult.

4.2 The Hydra model of interface state?

When Hercules fought the nine-headed Hydra he at first thought it easy, one by one he chopped off the venomous heads. But, from each bloody stump two new heads grew …

Early user-interface architecture models, in particular the Seeheim model [Pfaff, 1985], were monolithic. Driven partly by object-oriented programming, this was superceded by more component based models such MVC (Model-View–Controller) [Lewis, 1995]. The PAC (Presentation–Abstraction–Control) model [Coutaz, 1987] was developed partly to deal with the problematic dependencies between view and controller, but also in order to better model the hierarchical composition of components and sub-components within an interface. Similarly, MEAD captures aggregate objects in shared-state collaborative systems [Bentley, 1994].

However, there is also dynamic level of state decomposition within most interactive systems.

Imagine you are using a word-processor. There are immediately two distinct kinds of state: (i) the text and formatting of the document which will be saved; and (ii) the current selection, cut-paste buffer, etc. which will disappear when you exit the application.

Now suppose you open a tabbed dialogue box. There will be additional state connected with this dialogue box: (i) the formatting options you select typically do not have immediate effect, but instead are copies or ghost values that will overwrite the 'real' values only if the dialogue box is conformed; and (ii) the current selected tab or cursor position in a text box only have meaning during the life time of the box, as soon as you OK or cancel the box they are forgotten.

Notice the pattern: (i) a temporary copy of part of a deeper state (the file system, the current formatting); and (ii) some additional interaction state.

Suppose the dialogue box has a very long pull-down menu selection (perhaps font choice) that doesn't fot on the screen and so scrolls. Even this has a similar structure: (i) the currently selected menu item; and (ii) the current scroll position.

Now let's imagine we are editing some form of record either in a TP system as described in section 2 or in a web system. The original contents of the record are displayed and we edit them (with various levels of interaction state maintained by the web browser). There may be several screens related to the record, perhaps for multiple addresses, additional information. Possibly also when we 'submit' screens there may be validation errors. So, we have many web/TP transaction, but one higher-level user transaction 'edit the record'.

There is interaction state associated with this high level transaction, namely the temporary, part-edited contents of the record. This interaction state must be somehow passed on between individual web/TP transactions. In the case of single page records, this may often be managed without any explicit storage, just using the fact that the values are on screen and resubmitted with each transaction. For multi-page screens this is more complicated. To consider this we'll need to look more deeply at where systems store state.

4.3 State of the web

In the TP case study many of the original problems were due two things:

- understanding the issue of state
- recording and recalling the state

We'll assume that a better understanding of state may be engendered by appropriate education, appropriate techniques (such as scenario walkthroughs) and appropriate notations. This leaves the latter.

In the TP system there were various factors that simplified the problem:

- there was only one path of activity per user (no multiple windows)
- it was easy to identify the current user and terminal (given by the TP system)
- there was an easy way to store state associated with a particular terminal

Between them, these two allowed a specification style that assumed a single-threaded interaction per user and encoding of that.

In the web things are more difficult again!

Users may have multiple windows open, may 'duplicate' windows, may involve multiple frames. Furthermore the back and history functions allow users to revisit previous parts of the dialogue. So even when we think we understand the state of a dialogue we may find it spawning and duplicating itself in very strange manners. This is a big issue in itself and we won't deal with it further in this paper.

The web also has a variety of means whereby state information is preserved between user interactions.

(a) Some technologies preserve variables between interactions (e.g. JSP), but this has the same problems of accidental sharing as in the TP systems, so is only suitable for things like usage counters and cached values of persistent data.

(b) Data can be stored in various forms of persistent storage, sometimes in memory, more often in a database or other repository such as enterprise Java beans (EJB) in an enterprise server.
(c) Cookies can be set on a user's machine
(d) Hidden values can be set on web forms
(e) Links can be created on the web page with values encoded into the urls

The last two are equivalent in many ways, but the fact that you need to encode the same information differently in different context may well cause bugs!

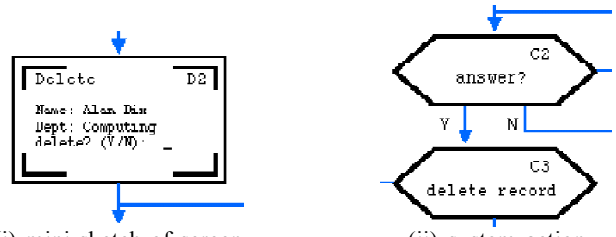
If we look at different web applications we find state stored in each of these ways, but in an ad hoc fashion, some in hidden variables, some in cookies. Many systems give support for session variables – data associated with a continuous period of use of the system (typically demarcated by some period of inactivity or through a login/logout). However, as we noted at the end of the previous section, one of the more common lifetimes for interaction state is the high-level user transaction, such as editing a record. This is shorter than a session, but for more complex interfaces cannot simply be carried with options (d) or (e). The typical progression seen in many systems is an initial coding of high-level transaction state in (d/e) followed by 'fixes' as this proves unsuitable, pushing it into session state, with further fixes to deal to deal with state that gets inappropriately stored too long!

The combination of a difficult distributed model, complex window management and incompatible versions of buggy browsers means that user interface development on the web will not be easy and this is reflected in the interfaces one encounters. However, understanding the nature of interface state does make this easier – both broad understanding of issues and more application specific knowledge based on using simple formalisms.

Looking forward there is clearly a need for the user interface community to investigate more appropriate formalisms, architectural frameworks and tools to support web and related interfaces.

5. Linking the formal and the informal

The flowcharts used in section 3.1 are composed of alternating user screens and system actions. Note again that the user screens are represented as a small sketch of the screen and the system actions are not complete, just a few words to give the reader an idea of what should happen. These seem like trivial things, but are crucial.



(i) mini-sketch of screen (ii) system action

Figure 4. main elemnts of flowchart

5.1 A picture tells …

An example I've used frequently when demonstrating formal methods in HCI is an extract from the instructions of a digital watch (figure 5). This is a form of state transition network for the watch – yes formal dialogue notation in a user manual! However, note also what makes it a very comprehensible notation for the user – little pictures of the watch represent the states!

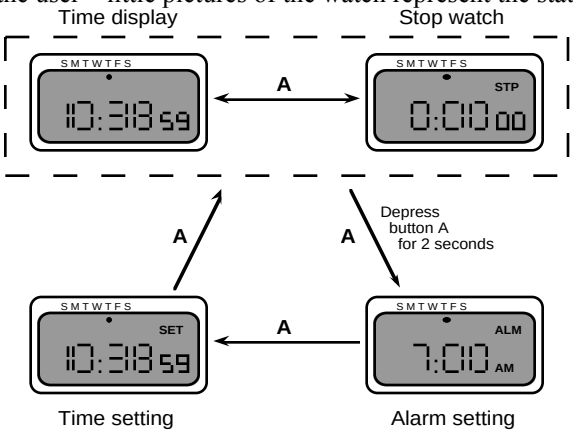


Figure 5 User instructions for wrist watch

Just imagine if the person producing the watch had instead decided to produce a more abstract STN (figure 6). Doesn't really work!

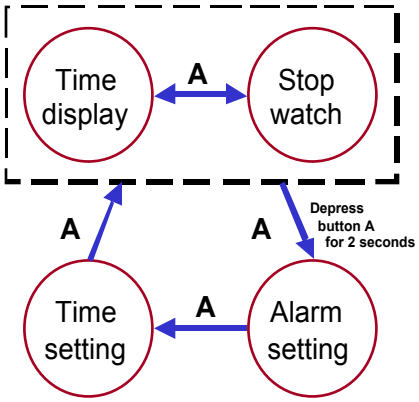


Figure 6 abstract instructions

This form of mini-screenshot is quite common in web-navigation design, certainly in paper-based design. In dialogue design tools it is more common to see simulated screens associated with 'playing' the formal description, for example, in Thimbleby's Hyperdoc [Thimbleby, 1993]. For micro-dialogue a thumbnail of the whole screen would be inappropriate as the critical features may be indiscriminable – the same problem also holds for automatically produce web page thumbnails. Note however, that the flowchart does not produce thumbnails of the (80x25) screen, but instead is a sketch, showing

enough details to make the different screens clear. For example, in screen D2 there would be far more employee details, but instead just enough are shown to give the general idea, but the crucial items is the prompt for 'Y/N'. Note also that screen D3 is shown with the extra reminder to distinguish it from D2. This is because it is a design notation, and intended to communicate between people as well as having a formal content.

5.2 Words too

Turning now to the system actions in figure 4.ii. As we've already discussed this deliberately doesn't give a great deal of detail. This is because the difficult area is in the large-scale dialogue structure not the internal details. The words used are sufficient to say what is required, but assume the developer knows how to program!

There is a temptation when working in formal domains to strive for completeness, in the sense of wanting to include everything within the formalism. However, the case study is a good reminder that formal precision is needed in some areas, but not everywhere.

This use of selective detail is more common among graphical formalisms, both in dialogue specification and task analysis, than in more 'mathematical-looking' notations, although this is no intrinsic reason for this.

ConcurTaskTrees (CTT) are an interesting example [Paternò, 2000]. They are related to CNUCE's LOTOS-based interactor model [Paternò, 1992]. The use of LOTOS implicitly encourages a much more detailed specification compared with the more top-down approach suggested by CTT. However, CTT is also like standard hierarchical task analysis [Shepherd, 1989]. In HTA, the decomposition of tasks is made precise but the relative ordering is left to less well formalised plans. CTT uses LOTOS operators to make this ordering precise, choosing an aspect of LOTOS that *complements* HTA. However, it still retains the important ability to leave sub-tasks undecomposed and then later refined. CTT has also been augmented by various tools that increase the pay back from the approach [Paternò, 1999].. So, CTT satisfies many of the criteria in section 3.2, and is enjoying a fair degree of external adoption compared with other user interface specification techniques.

5.3 But still formal

If you take the boxes in the flowchart and remove the annotations, just leaving the labels you get figure 7. You would never need to actually do this reduction, but this represents the part of the flowchart that can be analysed *without further human interpretation*. So, if one were to run the system and get the sequence of screens D1, D3, D2, one could tell, without knowing what input the user typed, that there was something wrong with the implementation if the dialogue.

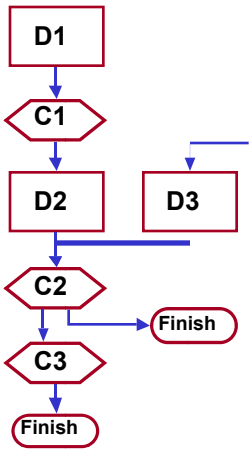


Figure 7 formal part of flowchart

It is possible to have diagrammatic notations for which it is not possible to extract formal properties. Furthermore, the formal interpretation of 'gaps' requires some care. For example, box C2 represnts the system examining the user input and making some choice. The actual choice depends on whether the user enters Y, N or something else. This is made clear in the human readable annotation, but in the strictly formal part of the flowchart this is non-deterministic, and without further information should be assumed to be demonically non-deterministic (that is always choosing the path you don't want!). In other words, one could not say definitively from figure 7 that it is possible to ever reach box C3, although this is evident reading (as a human) the full flowchart. This level of formal description is also suitable for automatic analysis such as the graph properties checked by Thimbleby [2001], or formal model checking.

Of course, one does not have to only verify the fully formal parts of a mixed formal-informal representation. Indeed, when the flowcharts were used in anger this was the case. During testing the flowchart could be used (as in formal testing regimes for programs), to determine all paths to test and and exhaustive test producer was performed. Although the formalism didn't 'know' which path through C2 should happen, the human tester did! Again, it is so easy to assume and all or nothing approach of completely automated or completely human analysis, whereas automatic analysis can be used most effectively to support human design.

6. Summary

The quite massive improvements made by formal dialogue specification in this case study show beyond doubt that formal methods can be of great value in user interface design … and in areas other than those it is usually assumed to cover.

Although this is an old example, the lessons are very pertinent. We haven't been able to discuss all of the lessons listed in section 3.2 in this paper, but there do seem to be many ways in which these can be applied.

The issue of state seems to be both generally difficult and also particularly difficult within user interfaces. Within web interfaces it begins to look well neigh intractable. Unfortunately, badly managed interface state is at best extremely confusing for users and at worst causes errors and bugs. Dealing with this through training, appropriate methods and appropriate support tools is therefore essential. Some suggestions are mode in this paper, for example, the use of rich scenarios alongside with formal state descriptions, this seems an area worthy of further study.

Perhaps the main failing of the formal methods community has been to be too precious about our formalisms. There is both a usability challenge (that is usability of formalisms) and a theoretical challenge in producing formalisms that can be used flexibly and profitably.

However, computer systems are the most formal systems produced by humankind, far more formal than diagrams or mathematics. So those eschewing formalism in HCI delude themselves. We all deal with formal methods in HCI, the ultimate challenge is to do it well.

References

Bentley R, Rodden T, Sawyer P, Sommerville, I. Architectural support for cooperative multi-user interfaces. IEEE COMPUTER special issue on CSCW, 1994; 27(5). pp 37-46.

E. M. Clarke, J. M. Wing (1996). Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, Vol. 28, No. 4, December 1996. pp. 626-643.

Coutaz J. PAC, an object oriented model for dialogue design. In: Bullinger H-J, Shackel, B. (eds) Human–Computer Interaction – INTERACT'87. Elsevier (North-Holland), 1987. pp 431-436.

Dix, A.J. and M.D. Harrison, Interactive systems design and formal development are incompatible?, in The Theory and Practice of Refinement, J. McDermid, Editor. 1989, Butterworth Scientific: p. 12-26.

Lewis The Art and Science of Smalltalk. Prentice Hall 1995.

Pfaff G, Hagen PJW. (eds) Seeheim Workshop on User Interface Management Systems. Springer-Verlag, Berlin 1985.

Palanque, P. and Paternò, F., editors (1997). Formal Methods in Human Computer Interaction. London, Springer-Verlag

Paternó, F. and G. Ballardin (1999). Model-Aided Remote Usability Evaluation. In Proceedings of Interact'99, pp. 434–442.

Paternó, F. and G. Faconti. On the use of LOTOS to describe graphical interaction. in Proceedings of HCI'92: People and Computers VII. 1992. Cambridge University Press. p. 155–173.

Paternò, F. (2000). Model-Based Design and Evaluation of Interactive Applications. London, Springer-Verlag

Shepherd. Analysis and training in information technology tasks. In D. Diaper, editor, Task Analysis for Human-Computer Interaction, chapter 1, pages 15-55. Ellis Horwood, Chichester, 1989

Thimbleby, H. W. (1993). Literate using for finite state machines. University of Stirling.

Thimbleby, H., P. Cairns, and M. Jones (2001). Usability Analysis with Markov Models. ACM Transactions on Computer-Human Interaction, Vol. 8, No. 2, June 2001, Pages 99–132.