Automatic critiques of interface modes

Jeremy Gow[1], Harold Thimbleby[2] and Paul Cairns[1]

[1] UCL Interaction Centre (UCLIC), University College London, 31-32 Alfred Place, London, WC1E 7DP, United Kingdom {j.gow, p.cairns}@ucl.ac.uk
[2] Department of Computer Science, University of Wales Swansea, Singleton Park, Swansea, SA2 8PP, United Kingdom h.thimbleby@swan.ac.uk

Abstract. We introduce a formal model of inconsistency-related mode confusion. This forms the basis of a heuristic methodology for critiquing user interfaces, using a matrix algebra approach to interface specification [11]. We also present a novel algorithm for automatically identifying modes in state-based interface designs, allowing a significant level of automated tool support for our methodology. The present paper extends and generalises our previous work on improving state-based interface designs [4].

Keywords: Mode, consistency, finite state machines, matrix algebra

Category: Formal methods in HCI.

1 Introduction

Modes are an integral part of user interface design — only the simplest of interfaces may be entirely 'modeless'. But modes are often the focus of criticism for interface evaluations, and in the wider HCI literature. A key question for interface designers is: how can we distinguish between good and bad modes during the design process?

The concept of mode is extremely general, and the term 'mode confusion' covers a variety of interface problems. A general definition by Leveson [8] is that "a mode is a set of mutually exclusive system behaviours" — the system behaves a particular way in one mode, but not in another — and that mode confusion arises from divergent user and system models. Much of the mode literature has addressed the problems caused by abstraction of system details in the interface, or the presence of automation, e.g. in autopilots [2, 9], and classically in text-editing [10]. However, in this paper we address what Leveson calls inconsistent behaviour: users may expect an action to have a consistent effect within a given mode, and any exceptions to this may be a source of user error. Consistency is important for users to correctly learn how an interface works by generalisation, and inconsistent behaviour is harder to learn.

Consistency is a commonly cited principle of interface design, but a somewhat nebulous one. We have previously proposed the identification and redesign of *partial behaviours* as a formal approach to consistent design [4], an approach that can be semi-automated to support the human designer/analyst. In this paper we generalise this work to a process of identifying *action modes* and ensuring *consistent relationships* between them. Our specific contributions are a new formal model of consistency-related mode confusion (Section 3), a methodology for identifying modes and potential mode confusions (Section 4), and a novel algorithm that automates this process (Section 5).

2 Modes and Consistency

We can see the relationship between consistency and modes more clearly if we take a state-based view, which is compatible with Leveson's general definition given above: a mode is defined as a *set of states* in which a user interface behaves in a particular way. Clearly, it follows that we may be in several modes at once, depending upon which behaviour we are considering.

Now suppose in mode M an action A typically exhibits a behaviour B. Following our definition, we can say there is another mode M' which is defined as 'the states in which A exhibits B'. The principle of consistency says that if we're in mode M then we should *always* be in mode M', that is, M should be subset of M'. If there are a few exceptional states of mode M that are not in mode M', then the interface behaves inconsistently.

Partial behaviours [4] give a formal description of a related consistency phenomena: an algebra of events (such as user actions) is used to specify behaviours that hold throughout an interface or in its specific modes. *Partial* behaviours are those algebraic properties that are *usually* true in a given mode, and indicate inconsistencies that may cause mode confusions. Recast in the mode terminology above, we can say that there is a given mode M, and another mode M' in which the 'partial' behaviour is true. The principle of consistency says that these modes should be equal, rather than almost equal.

Building on these ideas, we can characterise an 'interesting' (from a user's or designer's point of view) mode inconsistency as any relationship between modes that *almost* holds, and that the user is likely to perceive. That is, a simple relationship that holds for most states, but with a few exceptional states for which it does not. Clearly if it is true for only a few states then it is unlikely to be perceived; and if it is true for all states, there is no inconsistency. Provided the relation almost holds, experience is likely to lead the user to learn the relationship but not the exceptions, and their divergent user model can cause mode confusion. Moreover, a redesign of the user interface could make the relationship consistent and remove the source of errors. Figure 1 illustrates some inconsistent relationships between modes, and redesigns that enforce consistency.

3 A Model of Mode Inconsistency

As we only need consider mode relationships that the user is likely to (mis-)learn, we restrict ourselves to basic set theory. We have already discussed cases where

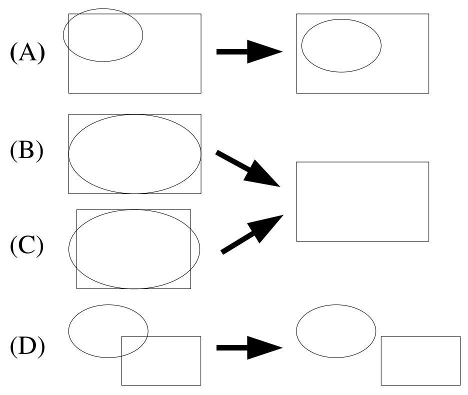

Fig. 1. Making mode relationships consistent. For two modes, represented by the ellipse (E) and the rectangle (R), four examples of inconsistent relationships are shown: (A) mode E is almost a subset of R; (B) mode R is almost a subset of E, and E *is* a subset of R; (C) modes E and R are almost subsets of each other; (D) E and R almost completely separate modes. In each case, the transformation to a consistent relationship is shown.

one mode is a subset of, or equal to, another. Other relationships may also be mislearnt, but in our experience they can be explained in terms of subsets ($\subseteq$), providing we use simple unions and complements of modes. For example:

- $A = B$ is equivalent to $A \subseteq B$ and $B \subseteq A$.
- A not intersecting with B is equivalent to $A \subseteq B^C$.
- $A_1, \ldots, A_n$ partitioning B is equivalent to $A_i \subseteq A_j^C$ for each $i \neq j$ and $A_1 \cup \ldots \cup A_n = B$.

In order to provide a model of mode inconsistency, we need to formalise the *almost subset* relation between sets of states.

3.1 Approximate and Near Subsets

In [4] we defined a partial behaviour as one that was true for a high proportion (called ρ below) of states within a mode. We take a similar, but more general, approach here. First, we define the the relation *approximate subset* $\underset{\approx}{\subseteq}$:

$$A \underset{\approx}{\subseteq} B \quad \leftrightarrow \quad \frac{|A \cap B|}{|A|} \geq \rho$$

This says that A is an approximate subset of B if the proportion of states in A that are also in B is at least ρ, where $0 \ll \rho < 1$, i.e. ρ is near 1. If the proportion is 1 then $A \subseteq B$ exactly.

The *nearly subset* relation $\underset{\sim}{\subset}$ is defined as an approximate subset excluding equality and actual subsets. Two modes A and B can be said to be inconsistent if and only if $A \underset{\sim}{\subset} B$ or $B \underset{\sim}{\subset} A$, where:

$$A \underset{\sim}{\subset} B \quad \leftrightarrow \quad \left(A \underset{\approx}{\subset} B \ \wedge \ A \nsubseteq B\right)$$

The definition of $\underset{\approx}{\subset}$ (and hence also $\underset{\sim}{\subset}$) is parameterised by the value ρ. Lowering ρ will increase the number of subsets which are classified as near subsets. In this paper, we take ρ to be a criterion, imposed by the designer. Instead, a design tool might determine a suitable ρ itself, or allow ρ to be dynamically adjusted by the designer; and, for example, sort near subsets by ρ for the attention of the designer. This practical question, while interesting for the design process, is irrelevant to our formal discussion.

3.2 Types of Mode

So far we have assumed that the user learns by experience the various interface modes and (possibly mislearns) the relationships between them. Our definition of mode — a set of states in which an interface behaves in a particular way — is deliberately general. However, the specific nature of the modes involved is likely to affect which modes and relationships are learnt by the user. In this paper we refine our model by distinguishing *action modes* and *indicator modes.*

Action modes are sets of states in which a particular combinations of user actions, or other events, have a consistent effect. Some of the action modes of a mobile phone might be the sets of states in which a cancel button returns the user to an initial state, or in which a combination of two buttons work as a toggle for a keypad lock, or in which a specific protocol is used to enter text. The mode is determined by the state transitions for the given events. A specific action mode is something the user believes, but cannot directly observe. That is, given the history of user actions up to this moment, the user *believes* (from experience or training) that the system is in an action mode. Note that each action mode is defined in terms of the system, not the user: a user may not understand the mode precisely, or even notice it exists! The aim here is to design systems that do not frustrate users that do notice them.

Indicator modes are sets of states in which the interface sends or displays consistent feedback to the user. For instance, the set of states for which a particular LED is lit, or for which music is played (e.g., this could be the 'play mode' on a MP3 player). The user can (in principle) *see* that the system is in a given indicator mode.

When defining compound modes using the union and complements of other modes, we make the simplifying assumption that modes are only combined with

Mode A	Mode B	Potential Error	Mode Error Type
Action	Action	Yes	In B when in A
Indicator	Action	Yes	In B when A observed
Action	Indicator	Yes	Not in A when B not observed
Indicator	Indicator	No	—

Table 1. Errors attributable to mode inconsistency $A \underset{\sim}{\subseteq} B$.

others of the same type. We assume that the user is unlikely to reason about modes defined as a combination of event behaviours and indicators.

Considering action and indicator modes, the effects of mode inconsistency $A \underset{\sim}{\subseteq} B$ depends on the combination of mode types involved. In general, it may be mislearnt as $A \subseteq B$. If the near-supermode B is an action mode, then the user may incorrectly believe that the action/effect association known to work in B will always work in mode A. In this case, the type of the near-submode A determines *when* this incorrect belief may be acted upon: if A is an indicator mode then this is whenever the given indicator is seen by the user; if A is an action mode then this will be whenever the user believes they are in A.

If B is an indicator mode then errors may be caused when the indicator is absent: when A is an action mode the user may assume that the absence means the interface is *not* in mode A. This error is only made in the exceptional states which are the cause of the mode inconsistency. No errors are caused by an inconsistency in which both A and B are indicator modes.

Table 1 shows a summary of the combinations of inconsistent modes that can mislead the user, and the type of mode errors in each case.

3.3 Using the Model

The model of mode inconsistency presented above is heuristic, because it does not suggest a specific consistent redesign, only a performance characteristic, namely the values of ρ and the counts of inconsistent modes. Furthermore, a more consistent redesign (i.e., fewer inconsistent modes or a larger ρ with the same modes) is not guaranteed to improve the interface: other design constraints may be violated — for example, redesign might replace three benevolently inconsistent modes with two atrociously inconsistent modes. However, redesign to reduce inconsistent modes has an underlying rationale, and there is an *expectation* that it will tend to improve the interface design other things being equal. Using a design tool, a designer could easily experiment with redesigns and the trade offs they represent, once problems are indicated.

4 A Method for Mode Analysis

Analysis requires the designer to already know which modes to look at. Indicator modes are part of the interface design, and hence need to be provided, at least implicitly, by the designer. Action modes are intrinsic to and implicit in the interface design and are therefore more difficult to find. In this section we provide a methodology for finding action modes. The process we describe here can be automated, and in the next section we provide an algorithm to do this.

Our approach is based on an algebraic specification of the interaction of states and user actions, as described in [11, 3]. Here we formally define an action mode as the largest set of states for which one of these algebraic properties is true. We also assume that the interface design is formally described as a finite state machine (FSM). Alternatively, it could be described in some higher-level formalism that can be translated into an FSM for the purposes of the mode analysis, such as OCDs [7], Promela [6] or Statecharts [5].

4.1 An Algebraic Model

We start with an FSM model of the user interface design, from which we generate a specification algebra. The FSM is defined as:

- A set of states S.
- A set of events E, including the actions available to the user.
- A transition relation $trans : S \times E \times S$.

An equivalent alternative formulation is to use a transition function $S \rightarrow powerset(S)$ instead. Also, we optionally have a set of indicator modes M_I.

A familiar FSM concept is the Boolean *transition matrix*, which defines all the transitions from every state to every other state. Instead, we define the *button matrix* for each event e as the transition matrix of the finite state machine restricted to transitions of event e. This is the basis of a very productive approach for a variety of user interface issues [11].

States are represented as row vectors, with each element corresponding to an individual state that the interface could be in. Hence, each particular state corresponds to a vector with a single non-zero entry, whereas groups of states — such as modes — may be represented as arbitrary vectors.

Formally, we use $||.||$ to denote the mapping from elements of the FSM to matrix algebra. So we distinguish between a state $s \in S$ and its vector $||s||$ and between an event $e \in E$ and its matrix $||E||$. Also, for any indicator mode $m \in M_I$ there is a vector $||m||$. Note that simulating the FSM corresponds to matrix multiplication: $trans(s, e, s')$ if and only if $||s||.||e|| = ||s'||$.

4.2 Specifying Modes

Figure 2 shows a BNF description of the full specification language for defining event/state properties. The simple semantics of the language is defined using matrix algebra: states (S) evaluate to vectors, events (E) evaluate to matrices, and there are simple calculable propositions (P) about them. We briefly and informally describe the features of the language here:

$$\begin{array}{lllllllllllllll}
\mathsf{P} ::= & \mathsf{E} \equiv \mathsf{E} & \mid & \mathsf{S} \equiv \mathsf{S} & \mid & \mathit{not}(\mathsf{P}) & \mid & \mathit{undo}(\mathsf{E}) \\
\mathsf{E} ::= & \mathit{Nothing} & \mid & \mathsf{e} & \mid & \mathsf{E.E} & \mid & \neg\mathsf{E} & \mid & \mathsf{E} \vee \mathsf{E} & \mid & \mathsf{E} \wedge \mathsf{E} & \mid & \mathit{go}(\mathsf{S}) \\
\mathsf{S} ::= & \mathit{All} & \mid & \mathit{None} & \mid & \mathsf{s} & \mid & \mathsf{S.E} & \mid & \neg\mathsf{S} & \mid & \mathsf{S} \vee \mathsf{S} & \mid & \mathsf{S} \wedge \mathsf{S}
\end{array}$$

Fig. 2. BNF grammar for algebraic specification of event/state properties P. Non-terminal E and terminal e are events, whereas S and s are states.

- Equivalence ($\equiv$) between either states or events. Defined as entry-wise equality between the corresponding vectors or matrices. $undo(E)$ holds if the matrix $||E||$ is invertible.
- $Nothing$ is an event which does nothing, which evaluates to the identity matrix.
- All and $None$ are the set of all states and no states, which evaluate to the vector with all non-zero elements and with all zero elements respectively.
- $E_1.E_2$ is the event E_1 followed by E_2. $S.E$ is the set of states reached from S via E. Both are evaluated by matrix multiplication.
- $go(S)$ is the event which goes from any state to the set of states S.
- $\neg S$ are the states not in S. Event $\neg E$ makes the transitions not taken by E.
- $S_1 \vee S_2$ are the states in S_1 and S_2. $E_1 \vee E_2$ is the event where either E_1 or E_2 occurs.
- $S_1 \wedge S_2$ are the states in S_1 and S_2. $E_1 \wedge E_2$ is an event which can be achieved by both E_1 and E_2.

The specification language is of limited expressiveness. In themselves these algebraic properties are not a sufficient basis for all the kinds of usability analysis we may want to do. However, they can be used to state and calculate simple behaviours of events and states. Moreover, the simplicity of the language allows us to easily construct properties in order to define action modes.

4.3 Mode Analysis

Mode analysis using the algebraic properties described above is a three stage process. Firstly, the interface designer must formulate properties that correspond to intended features of specific parts of the design. For example, they could specify that a cancel button returns the user to initial menu state $MainMenu$ and that along with the $*$ button it toggles the keypad lock:

$$Cancel \equiv go(MainMenu) \qquad Cancel.Star.Cancel.Star \equiv Nothing$$

For another device they might specify that a play button starts music playing, i.e. enters one of the set of states $Music$, and that the play and volume functions work independently:

$$Play \equiv go(Music) \qquad Play.(+Vol \vee -Vol) \equiv (+Vol \vee -Vol).Play$$

Each of these properties defines an action mode, i.e. the set of states in which the property is true. Although formulating them is a skilled task they are essentially quite simple, and within the capabilities of many interface designers given appropriate tool support. Indeed, they can also generated automatically for the designer to review using the algorithm described in the following section.

Secondly, given a set M_A of action modes, each defined by an algebraic property, a revised set of modes M'_A is formed by taking complements, unions and intersections of the modes in M_A. The same process is repeated for the indicator modes M_I, to give a set M'_I. Finally, pairs of modes from $M'_A \cup M'_I$ are compared using the $\subsetsim$ relation, avoiding pairs of indicator modes. Any inconsistent pairs should be considered by the designer as candidates for redesign.

5 Automating the Method

To support the designer in applying our mode consistency methodology, we now present a novel algorithm that can automatically find algebraic properties/action modes to 'feed into' the analysis. The algorithm can also be used to generate action specifications as part of a more general design methodology [11]. We have developed MAUI, a Java/XML-based prototype design tool that supports specification with, and automatic generation of, such properties [3]. A prototype implementation of this mode analysis technique has been made in MAUI, which we intend to use for an evaluation of this methodology. Early results suggest it can handle realistic interface designs.

The algorithm produces a restricted subset of specification properties which are useful for defining action modes. Specifically, those of the forms:

$$A_1 . \ldots . A_n = B_1 . \ldots . B_m \qquad\qquad C_1 . \ldots . C_p = go(s)$$

for $A_i, B_i, C_i \in E$ and $s \in S$. It generates these properties up to a bound N on the maximum product length (i.e. on n, m and p), which is set by the designer. The algorithm is exponential in N, but we have found practical examples can easily be computed. It would be possible to implement this process using more sophisticated techniques, e.g. [1], should computational resources become a problem.

5.1 Constructing Equivalence Classes

Given the set of events E we want to combine individual events into composite actions, represented by the set of terms $\mathcal{T}_E$ formed by matrix multiplication. This is an infinite set, and we limit ourselves to investigating a finite subset $\mathcal{T}_E^N$ (for some $N > 0$), defined inductively as:

$$\frac{e \in E}{e \in \mathcal{T}_E^1} \qquad \frac{t \in \mathcal{T}_E^k \quad e \in E}{t.e \in \mathcal{T}_E^{k+1}}$$

The first step is to partition $\mathcal{T}_E^N$ into a set of equivalence classes $\mathcal{C}_E^N$, defined by equivalence between events. Formally $\mathcal{C}_E^N = \{C_1, \ldots, C_m\}$ such that $\bigcup C_i = \mathcal{T}_E^N$

```
classify(term t) {
  if (exists c in EQC and matrix(rep(c)) = matrix(t))
      put t in c;
  else
      new class c';
      put c' in EQC;
      if (matrix(t) = matrix(go(s)) for state set s)
          put go(s) in c';
          r = go(s);
      else
          r = t;
      fi
      put t in c';
      extend rep so that rep(c') = r;
      put r in Novel;
  fi
}
```

Fig. 3. The `classify` function.

```
redundant(term t):bool {
  red = false;
  for (c in EQC)
      for (q in c)
          if (not(q == rep(c)) and t == pq) red = true;
  return red;
}
```

Fig. 4. The `redundant` function.

and for $a \in C_i$ and $b \in C_j$, $||a|| = ||b||$ iff $i = j$. There is also a *representative* function $\phi : \mathcal{C}_E^N \to \mathcal{T}_E^N$ which selects a distinguished element from an equivalence class, i.e. $\phi(C) \in C$.

Computing the equivalence classes involves the gradual construction of $\mathcal{C}_E^N$ and ϕ, a process we describe here in pseudo-code, to give an clear description of the algorithm. An actual implementation will be able to make numerous efficiency savings over the 'code' here (at the expense of being more obscure). The computation uses various global data structures: As inputs, the set of events `E` and a function `matrix` from terms to matrices; used during the computation are the set of novel terms `Novel` and the set of unique events `Unique`; As outputs, the set of equivalence classes `EQC` and the function `rep` which returns the representative element $\phi(C)$ for each class C.

The main algorithm depends on three functions: `classify` is used to place a new term in an appropriate equivalence class (see Figure 3); `redundant` tests whether a term has a subterm that has already been computed (see Figure 4); and `newTerms` computes the new terms introduced to $\mathcal{C}_E^N$ when n is increased by one (see Figure 5). The algorithm that computes $\mathcal{C}_E^N$ for $N > 1$ is shown in Figure 6. It calls `newterms` for successive values of n, until the bound is reached

```
newTerms(int n) {
  Seeds = Novel;
  Novel = {};
  for (t in Seeds)
      for (e in Unique)
          if (not(redundant(t.e))) classify(t.e);
}
```

Fig. 5. The `newTerms` function.

```
computeEquivClasses(int n) {
  C = {Id};
  r(C) = Id;
  EQC = {C};
  Novel = {};
  for (e in E)
     classify(e);
  Unique = Novel;
  if (n > 1)
      i = 1;
      do
         newTerms(i);
         i++;
      until
         (i > n) or Novel = {};
}
```

Fig. 6. The main algorithm for computing equivalence classes.

or until no new novel terms are introduced by this cycle. In the latter case we say the equivalence classes are *saturated*: the classification for any larger term can be computed algebraically from the existing classification, and so further classification is pointless. Note that the algorithm is presented in a simple form, and considerable efficiency savings could be made, e.g. by using a dynamic programming approach to build up terms during the classification.

5.2 Identifying modes

Having computed the equivalence classes $\mathcal{C}_E^N$ for the set of terms T_E^N, we can generate algebraic properties that define action modes. Global properties of the interface can be found be equating a term with its class representative, but here we are interested in the non-global properties which define modes. Formally, pairs of equivalence classes C_1, C_2 are compared to see if there is a set of states m such that

$$go(m).\phi(C_1) \quad \equiv \quad go(m).\phi(C_2)$$

i.e. that matrices for C_1 and C_2 are equivalent for the rows corresponding to the states of m. If this holds, then m is an action mode defined by the property $\phi(C_1) \equiv \phi(C_2)$.

This process may return unmanageably many modes for any non-trivial interface, and the potential modes must be pruned in some way: a minimum mode size for m can be enforced, or a bound on size of property (we treat repetitions of the same event as a single event, e.g. $e.e$ is e^2.) Another technique is to generalise a set of similar modes into a single mode, e.g. $e \equiv f$ for mode m_1, $e.e \equiv f$ for m_2 and $e.e.e \equiv f$ for m_3 can be generalised to $\exists N.\ e^N \equiv f$ for mode $m_1 \wedge m_2 \wedge m_3$. Finally, the designer may review the generated modes to select those they judge useful for further analysis.

6 Further Work

Our priority for further research is the evaluation of the methodology described here, using the prototype implementation in MAUI. This is initially being done by testing MAUI on a corpus of interface designs, which will lead on to further experiments with non-expert designers. We would also like to refine our model of mode inconsistency to e.g. account for more specific mode types. There are other areas of mode confusion which might benefit from more formal models that complement existing heuristic approaches, such as [8].

Modifying a design is a rather open-ended activity and is of course best suited to human designers. A tool can however still make very useful suggestions for reducing mode inconsistencies. We intend to extend our techniques to suggest possible mode redesigns as well as improving the identification of existing mode inconsistencies. A design tool can establish or check many other sorts of property. Our tool, for example, checks states are strongly connected — since any state in another component is unusable and merely represents unnecessary implementation complexity. There are some devices (e.g., a single use fire extinguisher) where strong connectivity is unwanted, at least for certain states. There are many other interesting kinds of property, and currently our tool can only specify a limited range. Further work should explore convenient ways to express a large class of such properties — and in a way that is 'designer friendly' rather than logic-driven.

We have various parameters to our methodology as constants: the minimum bound for inconsistency ρ, the maximum product size N, and the minimum mode size. These could, instead, be 'sliders' on the design tool: the designer could adjust the slider to get the level of detail (pedantry?) that they want to work with. Perhaps — and we look forward to trying this — it would be insightful to visualise the effects of these parameters by drawing graphs. Further research may develop ways to automatically determine values that give a 'reasonable' description of the interface design.

Our notion of inconsistency could be refined by measuring near-subsets based on a weighting of interface states. As users will spend more time in some states, these will bias their perception of interface behaviour, and therefore their learning (and mislearning) of mode relationships.

7 Conclusions

We have presented a general, formal model of inconsistency-related mode confusion, and an accompanying methodology for mode analysis, a significant amount of which can be automated. The model is general enough to provide a framework for further study into consistency and mode confusion.

We have argued here that inconsistent modes are an important user interface feature that are likely to cause users problems. Generally, they should best be avoided unless there are contra-indications due to the nature of the user's task. Unfortunately inconsistent modes are a non-trivial implicit feature of user interfaces, and therefore not easily avoided by diligent designers; fortunately, as this paper showed, they can be enumerated automatically by appropriate tools, such as the MAUI system, which we are in the process of evaluating.

A huge advantage of our approach is that it gives the ordinary designer useful usability insights easily; a simple specification language is used, and with appropriate tool support we anticipate that little training required to use the approach — most of which is fully automated.

Acknowledgements Jeremy Gow is funded by EPSRC grant GR/S73723/01. Harold Thimbleby is a Royal Society Wolfson Research Merit Award Holder.

References

1. E. M. Clarke, E. A. Emmerson & A. P. Sistla (1999), *Model Checking*. MIT Press.
2. A. Degani (1996), "Modelling human-machine systems: On modes, error and patterns of interaction." PhD thesis, Georgia Institute of Technology.
3. J. Gow & H. Thimbleby (2004), "MAUI: An interface design tool based on matrix algebra." In R. Jacob, Q. Limbourg & J. Vanderdonckt (eds), *Proc. 4th International Conference on Computer-Aided Design of User Interfaces*. Kluwer.
4. J. Gow, H. Thimbleby & P. Cairns (2004), "Misleading behaviour in interactive systems," In A. Dearden & L. Watts (eds), *Proceedings of the 18th British HCI Group Annual Conference (HCI 2004)*, Volume 2.
5. D. Harel & A. Naamad (1996), "The STATEMATE semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, **5**(4):293-333.
6. G. J. Holzmann (2003), "The SPIN model checker," Addison-Wesley.
7. D.-S. Lee & W. C. Yoon (2004), "Coupling structural & functional models for interaction design," *Interacting with Computers*, **16**:133–161.
8. N. G. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga & J. D. Reese (1997), "Analyzing software specifications for mode confusion potential." In C. W. Johnson (ed.) *Proc. Workshop on Human Error and System Development*, Glasgow, Scotland, pp132–146.
9. J. Rushby (2002), "Using model checking to help discover mode confusions & other automation surprises," *Reliability Engineering & System Safety*, **75**(2):167–177.
10. H. Thimbleby (1982), "Character level ambiguity: Consequences for user interface design," *International Journal of Man-Machine Studies*, **16**:211–225.
11. H. Thimbleby (2004), "User interface design with matrix algebra," *ACM Transactions on Computer-Human Interaction*, **11**(2):181–236.