

Automatic Critiques of Interface Modes

Jeremy Gow¹, Harold Thimbleby², and Paul Cairns¹

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 [12]. 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 generalises our previous work on improving state-based interface designs [5].

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 [3, 10], and classically in text-editing [11]. 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. Miller and Potts emphasise the need for a formal definition of mode inconsistency [9]. We have previously proposed the identification and redesign of *partial behaviours* as a formal approach to consistent design [5], 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 (Sect. 3), a methodology for identifying modes and potential mode confusions (Sect. 4), and a novel algorithm that automates this process (Sect. 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 are 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 is inconsistent.

Partial behaviours give a formal description of a related consistency phenomena [5]: an algebra of events, including user actions, is used to specify interface behaviour. A *partial* behaviour is an algebraic property that is *usually but not always* true in a given mode—it indicates an inconsistency that may cause mode confusions. The methodology recommends redesign so that the property holds consistently throughout the mode.

This paper generalises our previous work on partial behaviours. 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

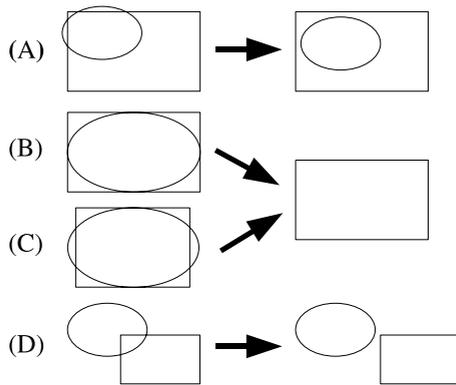


Fig. 1. Making mode relationships consistent. For two modes, represented by the ellipse E and the rectangle R , four examples of inconsistent relationships are shown: (A) mode E is almost a subset of R ; (B) mode R is almost a subset of E , and E is a subset of R ; (C) modes E and R are almost subsets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

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 one mode is a subset of, or equal to, another. Other relationships may also be mislearned, but in our experience they can be explained in terms of subsets (\subseteq), providing we also 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, \dots, A_n partitioning B is equivalent to $A_i \subseteq A_j^C$ for each $i \neq j$ and $A_1 \cup \dots \cup A_n = B$.

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 [5], 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* $\underset{\approx}{\subseteq}$:

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

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 $\underset{\subset}{\subseteq}$ 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 $A \underset{\subset}{\subseteq} B$ or $B \underset{\subset}{\subseteq} A$, where:

$$A \underset{\subset}{\subseteq} B \leftrightarrow (A \underset{\approx}{\subseteq} B \wedge A \not\subseteq B)$$

The definition of $\underset{\subset}{\subseteq}$ (and hence also $\underset{\approx}{\subseteq}$) 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 Action vs Indicator Modes

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 particular combinations of user actions, or other events, have a consistent effect. For a mobile phone this could 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. This belief may inform the subsequent interaction. 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 these modes.

The second type of mode distinguished here are indicator modes. These 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.

The effects of mode inconsistency $A \subsetneq B$ depends on the combination of mode types involved, as summarised in Table 1. In general, $A \subsetneq B$ 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 action mode then this is whenever the user believes they are in A ; if A is an indicator mode then it will be whenever the given indicator is observed.

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. Types of user-interface divergence attributable to mode inconsistency $A \subsetneq B$

Mode A	Mode B	User belief	Interface state
Action	Action	In B	In A and not in B
Indicator	Action	In B	A observed and not in B
Action	Indicator	Not in A	In A and B not observed
Indicator	Indicator	—	—

When defining compound modes via union and complement, we make the simplifying assumption that modes are only combined with others of the same type, with the combined mode being of that type. For example, the union of two indicator modes is an indicator mode. We speculate that users are less likely to reason about modes defined as a combination of event behaviours and indicators.

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 examine. 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 [12, 4]. 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 translated into an FSM from a higher-level formalism, perhaps one of many standard state-based notations for system specification, such as Statecharts [6] or Promela [7].

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 formulation could 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 [12].

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 $||\cdot||$ 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 $||s|| \cdot ||e|| = ||s'||$.

4.2 Specifying Action Modes

We describe the specification language for defining event/state properties (and hence action modes) using standard BNF notation:

$$\begin{aligned} P ::= & E \equiv E \quad | \quad S \equiv S \quad | \quad not(P) \quad | \quad undo(E) \\ E ::= & Nothing \quad | \quad e \quad | \quad E.E \quad | \quad \neg E \quad | \quad E \vee E \quad | \quad E \wedge E \quad | \quad go(S) \\ S ::= & All \quad | \quad None \quad | \quad s \quad | \quad S.E \quad | \quad \neg S \quad | \quad S \vee S \quad | \quad S \wedge S \end{aligned}$$

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. The language operators correspond to simple vector/matrix operations. Equivalence (\equiv) is between either states or events, defined as entry-wise equality between the vectors/matrices. The proposition $undo(E)$ holds if the matrix $||E||$ is invertible. *Nothing* is an event which does nothing, and 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. The event $go(S)$ takes any state to the set of states S .

The remainder of the operators are Boolean: $\neg S$, $S_1 \vee S_2$ and $S_1 \wedge S_2$ are respectively the states not in S , those in S_1 or S_2 and those in S_1 and S_2 . Event $\neg E$ causes only those transitions not taken by E . Event $E_1 \vee E_2$ represents transitions caused by E_1 or E_2 (their combined functionality), while $E_1 \wedge E_2$ represents those caused by both (their common functionality).

This specification language is of limited expressiveness. In themselves the 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 that define action modes.

4.3 Mode Analysis

Mode analysis using these algebraic properties is a three stage process. First, 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:

$$\text{Cancel} \equiv \text{go}(\text{MainMenu}) \quad \text{Cancel}.\text{Star}.\text{Cancel}.\text{Star} \equiv \text{Nothing}$$

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:

$$\text{Play} \equiv \text{go}(\text{Music}) \quad \text{Play}.(+\text{Vol} \vee -\text{Vol}) \equiv (+\text{Vol} \vee -\text{Vol}).\text{Play}$$

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 be generated automatically for the designer to review using the algorithm described in the following section.

Given a set M_A of action modes, each defined by an algebraic property, the next stage is to form a revised set of modes M'_A 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 \subsetneq relation, avoiding pairs of indicator modes. Any inconsistent pairs are flagged 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 [12]. We have developed MAUI, a Java/XML prototype design tool that supports specification with and automatic generation of such properties [4]. 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 can only generate a restricted subset of specification properties. Specifically, those of the forms:

$$A_1 \dots A_n = B_1 \dots B_m \quad C_1 \dots C_p = \text{go}(s)$$

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. A more sophisticated implementation, e.g. [2], could be used should computational resources become a problem.

5.1 Automation—Finding Equivalences

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:

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

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 $\mathcal{C}_E^N = \{C_1, \dots, C_m\}$ such that $\bigcup C_i = \mathcal{T}_E^N$ and for $a \in C_i$ and $b \in C_j$, $\|a\| = \|b\|$ iff $i = j$. There is also a *representative* function $\phi : \mathcal{C}_E^N \rightarrow \mathcal{T}_E^N$ 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 a 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). For brevity, our code uses various global data structures: the set of events E , a function `matrix` from terms to matrices (both given as inputs), the set of novel terms `Novel`, the set of unique events `Unique`, the set of equivalence classes `EQC` and a function `rep` which maps each class C to its representative element $\phi(C)$. The purpose of the algorithm is to compute `EQC` and `rep`.

The main algorithm depends on three functions: `classify` is used to place a new term in an appropriate equivalence class (see Figure 2); `redundant` tests whether a term has a subterm that has already been computed (see Figure 3); and `newTerms` computes the new terms introduced to \mathcal{C}_E^N when n is increased by one (see Figure 4). The algorithm that computes \mathcal{C}_E^N for $N > 1$ is shown in Figure 5. It calls `newterms`

```

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. 2. 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. 3. The redundant function

```

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

```

Fig. 4. 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. 5. The main algorithm for computing equivalence classes

for successive values of n , until the bound is reached 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 Automation—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 by equating a term with its class representative, but here we are interested in the non-global properties that 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

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

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 should be able to review and filter the generated modes to select those they judge useful for further analysis.

6 Discussion

Several authors have used formal methods to model mode problems, in particular those caused by abstraction and automation [3]. Rushby used model checking to find divergence between user and system models [10]. In a similar vein, Bredereke and Lanke-nau provided a more rigorous characterisation of mode confusion [1]. In contrast, our model addresses a different form of mode confusion — specifically caused by inconsistency — and does not require a user model. Other types of mode confusion might be better understood with similar formal models that complement existing heuristic approaches, such as [8].

Inconsistency-related mode confusion is a familiar heuristic concept in HCI. Miller and Potts take a formal approach to avoiding such mode confusions in the context of cockpit interfaces [9]. They prove lemmas in PVS stating that certain switches always act as toggles. In our approach this is expressed as a simple algebraic property, e.g. $A.A = \text{Nothing}$. Not only is this a clearer representation, we also formally characterise inconsistent violations of this property, as well as many others. Moreover, we have provided a method for automating this analysis.

The scalability of automating our approach is an issue: the ‘combinatorial explosion’ will render our automatic analysis impractical for large enough systems. As discussed above, this can be mitigated with more sophisticated implementations. Comparable limits in model checking have been pushed back so that extremely large state systems can now be analysed. More importantly, even a restricted amount of system analysis may reveal interesting mode inconsistencies.

Our priority for further research is the evaluation of the theory and methodology described here, using the prototype implementation in MAUI [4]. This is initially being done by testing the theory on a corpus of interface designs. 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. There is also scope for extending the classification of mode types beyond action/indicator.

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. Further work should explore convenient ways to express a large range 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.

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 shows, they can be enumerated automatically by appropriate tools, such as the MAUI system, which we are in the process of evaluating.

The approach has the potential to give ordinary designers useful usability insights: a simple specification language is used and, with appropriate tool support and automation, we anticipate that little training would be required.

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

References

1. J. Brederke & A. Lankenau (2002), “A rigorous view of mode confusion.” In Anderson, Bologna & Felici (eds), *Computer Safety, Reliability and Security, Proc. SAFECOMP 2002*. LNCS 2434, Springer, pp19–31.
2. E. M. Clarke, E. A. Emmerson & A. P. Sistla (1999), *Model Checking*. MIT Press.
3. A. Degani (1996), “Modelling human-machine systems: On modes, error and patterns of interaction.” PhD thesis, Georgia Institute of Technology.
4. J. Gow & H. Thimbleby (2004), “MAUI: An interface design tool based on matrix algebra.” In Jacob, Limbourg & Vanderdonck (eds), *Computer-Aided Design of User Interfaces IV, Proc. CADUI 2004*. Kluwer.

5. J. Gow, H. Thimbleby & P. Cairns (2004), "Misleading behaviour in interactive systems," In Dearden & Watts (eds), *Proc. 18th British HCI Group Annual Conference (HCI 2004)*, Volume 2.
6. D. Harel & A. Naamad (1996), "The STATEMATE semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, **5**(4):293-333.
7. G. J. Holzmann (2003), "The SPIN model checker," Addison-Wesley.
8. N. G. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga & J. D. Reese (1997), "Analyzing software specifications for mode confusion potential." In Johnson (ed.) *Proc. Workshop on Human Error & System Development*, Glasgow, pp132-146.
9. S. P. Miller & J. N. Potts (1999), *Detecting Mode Confusion Through Formal Modeling and Analysis*. NASA Contractor Report, NASA/CR-1999-208971.
10. J. Rushby (2002), "Using model checking to help discover mode confusions & other automation surprises," *Reliability Engineering & System Safety*, **75**(2):167-177.
11. H. Thimbleby (1982), "Character level ambiguity: Consequences for user interface design," *International Journal of Man-Machine Studies*, **16**:211-225.
12. H. Thimbleby (2004), "User interface design with matrix algebra," *ACM Transactions on Computer-Human Interaction*, **11**(2):181-236.