Systematic Analysis of Control Panel Interfaces Using Formal Tools

J. Creissac Campos[1] and M. D. Harrison[2]

[1] Department of Informatics/CCTC, Universidade do Minho, Braga, Portugal
Jose.Campos@di.uminho.pt
[2] School of Computing Science, Newcastle University, UK
Michael.Harrison@ncl.ac.uk

Abstract. The paper explores the role that formal modeling may play in aiding the visualization and implementation of usability requirements of a control panel. We propose that this form of analysis should become a systematic and routine aspect of the development of such interfaces. We use a notation for describing the interface that is convenient to use by software engineers, and describe a set of tools designed to make the process systematic and exhaustive.

1 Introduction

Applying formal techniques to analyze interactive systems makes possible a more systematic approach to the evaluation of the usability of a new design. Formal techniques can provide an incisive analysis that is effective in uncovering potential unforeseen interaction problems which can then be explored from a usability perspective. The paper demonstrates how a collection of tool supported property patterns (akin to those described in [12]) can be used to make this process more systematic. The interface under analysis is specified using Modal Action Logic (MAL) which focuses on the meaning and effect of action. The approach is illustrated by analyzing the air conditioning system for a family car. In addition to potential usability problems, the patterns help discover discrepancies between assumed meanings based on the user manual and meanings derived by experimenting with the system.

The proposed techniques are similar in aim to those of [5] and [14]. MAUI [9] is a comparable tool supported technique for analyzing control panel systems. The work presented here differs by (1) supporting a textual design specification notation and (2) supporting the systematic analysis of a set of standard interface properties. There is no space in this paper to do full justice to a comparison between these techniques and to compare the range of other techniques that have been developed recently, see for example [11] for a review. The focus here is to demonstrate how formal techniques can be made more routine and systematic through a real example. The example illustrates techniques that fit naturally with the programmer's view of the system while at the same time triggering a usability perspective. The paper describes:

1. a notation that clearly and simply captures characteristics of interactive devices
2. a set of properties that can be systematically checked of the interactive system
3. a tool that pulls together the means of specification and the means of checking, that is accessible to appropriate developers.

Finally, discovery tools are required to explore the consequences of the problems uncovered by these techniques. The systematic approach is supported by the IVY tool developed to check MAL specifications. The paper explains the characteristics of the tool and comments on how the formal approach can be complemented by a more user focussed analysis.

2 The Example

The example is the automatic air conditioning panel of the Toyota Corolla (2001 European version). The actions of the air conditioning system concern setting temperature and altering the rate and direction of the flow of air. While the actions associated with temperature and rate of flow are relatively straightforward, complications involve the number of modes that deal with the direction of flow. The complete set of actions and displays is identified below.

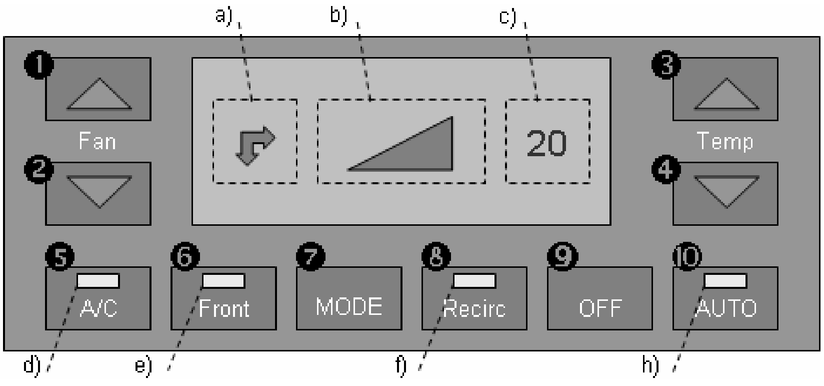


Fig. 1. The air-conditioning control panel

Figure 1 shows what the control panel looks like. The panel has ten buttons (these are enumerated in the figure) and there are seven display features that can change through use of the air conditioning system ((a)-(h)). These elements are first identified before describing them in more detail through the specification. The buttons correspond to actions in the model, the names of the actions are as follows: (1) increase fan speed (*fanspeedup*); (2) decrease fan speed (*fanspeeddown*); (3) increase target temperature (*tempup*); (4) decrease target temperature (*tempdown*); (5) select air conditioning mode (*ackey*); (6) select windscreen (front) flow mode (*frontkey*); (7) select flow mode (*modekey*); (8) select air intake mode (*airintakekey*); (9) off (*off*); (10) select automatic mode (*autokey*).

The displayed indicators are perceivable attributes of the state. These are identified in the model by the names in brackets in the following list: (a) flow mode (*airflow*); (b) fan speed (*fanspeed*); (c) target temperature (*settemp*); (d) air-conditioning on/off (*ac*); (e) wind screen (front) flow mode on/off (*front*); (f) recirculation air intake mode (*airintakefresh*); (g) automatic mode on/off (*auto*).

3 The Modeling Notation

A MAL specification is produced focusing on relevant actions and attributes of the state. The semantics of MAL is discussed in more detail in [6,1]. This set of actions and attributes may be modified as additional assumptions about the specification are identified through experimenting with the system or exploring properties of the specification. The specification is structured using hierarchical interface components (called interactors). In the example one interactor describes all the actions and visible attributes of the state of the system. No assumptions are made in this analysis about other properties that may be important from a usability point of view. For example, a user may feel or hear the effect of changes in the temperature, fan speed and where the air is flowing. These additional modalities are ignored. Context effects, for example whether the car windows are open or not, are also ignored. In practice these aspects of the system could be considered additionally if appropriate.

There are three types of MAL axioms. *Propositional axioms* describe invariants over the state of the interactor. *Modal axioms* describe effects of an action in terms of the state of the interactor. The modal axioms describe production rules that define a state machine. Finally *deontic axioms,* which are not used in this example, capture conditions that determine when actions are permitted or obligatory.

Three visible state attributes are important to the functioning of the air conditioning system: temperature (*settemp*), flow speed defined by the fan speed indicator (*fanspeed*) and flow mode (*airflow*) that defines where the air flows, for example dashboard level or floor level or to the windscreen. These attributes, see (a)-(c) in Figure 1, can be described as follows:

interactor *main*
  attributes
  [vis] *settemp* : *Temp*
  [vis] *airflow* : *AirFlow*
  [vis] *fanspeed* : *FanSpeed*

The specification consists of one interactor named *main*. The modality [vis] of each attribute is “visible”. These attributes of the states are changed by three sets of buttons: *settemp* by [*tempup*] (3) and [*tempdown*] (4); *fanspeed* by [*fanspeedup*] (1) and [*fanspeeddown*] (2); the flow mode is controlled by a more complicated set of buttons. While the manual provided an initial explanation of how the controls are used, this information was updated in the light of analysis and experimentation.

$$[tempup]\ (settemp < MAXHOT \rightarrow settemp' = settemp + 1) \\ \wedge (settemp = MAXHOT \rightarrow keep(settemp))$$
$$[tempdown]\ (settemp > MAXCOLD \rightarrow settemp' = settemp - 1) \\ \wedge (settemp = MAXCOLD \rightarrow keep(settemp))$$

Normal logical operators are used in the specification; actions appear in square brackets. The expression to the right of the action describes how the state attributes are changed. In the case of [*tempup*] if temperature is lower than the maximum possible (*MAXHOT*) it is incremented. The new state of *settemp* is indicated by priming the attribute, hence (*settemp’*) becomes the previous plus one. If temperature is already equal to *MAXHOT*, then its value does not change: (*keep*(*settemp*)). If an attribute does not appear in the keep list and its behaviour is not defined by the axioms, then it assumes a random value. [*tempdown*], [*fanspeedup*] and [*fanspeeddown*] have similar definitions. More axioms are required for actions associated with where the air flows. Possible air flow modes are defined by the set:

$AirFlow = \{panel, double, floor, floorws, wsclear\}$.

Whether the air conditioning system (temperature, fan and airflow) is switched on or off is not yet captured in these axioms. The fact that this aspect of the design is not clearly visible in the system is the reason for this omission. The only possible indicator is the fan speed (see indicator (b) in Figure 1), but this is an indirect and not very salient association. This omission raises an issue for the designer as to whether this aspect of the design should be made more clear.

The air conditioning mode selector key (5) is defined when the system is on and when it is off. When off, pressing the button has no effect on the state attributes, when on the mode key simply changes the *ac* attribute, toggling its value.

$$on \rightarrow [ackey]\ ac' = \neg ac \wedge keep(auto, airintake, settemp, on, front, airflow, fanspeed)$$
$$\neg on \rightarrow [ackey]\ keep(auto, airintake, settemp, on, front, airflow, fanspeed, ac)$$

The windscreen (*flow*) mode selection button [*frontkey*] has the following axioms:

$$on \rightarrow [frontkey]\ on' \wedge front' = \neg front \wedge keep(settemp)$$
$$\neg on \rightarrow [frontkey]\ on' \wedge front' \wedge keep(settemp)$$
$$[frontkey]\ front' \rightarrow (\neg auto' \wedge \neg airintake' \wedge ac')$$
$$front \leftrightarrow airflow = wsclear$$

Hence when the system is on, pressing the front button will toggle the front attribute, and when switched off the button will switch it on (*on'* asserts the new value of *on* is *true*). The final axiom specifies an invariant, namely when the front mode is set the airflow is always in windscreen clear mode. The *modekey* and *airintakekey* are specified as follows:

$$[modekey]\ \neg auto' \wedge front' \wedge keep(airintake, settemp, on, fanspeed)$$
$$\begin{aligned}\neg front \rightarrow [modekey]\ & (airflow = panel \rightarrow airflow' = double) \\ & \wedge (airflow = double \rightarrow airflow' = floor) \\ & \wedge (airflow = floor \rightarrow airflow' = floorws) \\ & \wedge (airflow = floorws \rightarrow airflow' = panel) \wedge keep(ac)\end{aligned}$$
$$\begin{aligned}[airintakekey]\ & airintake' = \neg airintake \\ & \wedge keep(auto, settemp, on, front, airflow, fanspeed, ac)\end{aligned}$$

It was difficult to produce an unambiguous and accurate specification of this system based on *both* the manual and use of the system because: (a) the manual is not clear in places – e.g., “When the Front key is pressed, air flows mainly through the windscreen vents, and the FRESH air intake mode is automatically set” is only true when the front mode is off; (b) the manual is incomplete - e.g., the fact that pressing the mode key in auto mode turns the mode indicator off is not described in the manual; (c) the manual is inconsistent with the device - e.g., references to the A/C button being depressed are not consistent with the actual user interface where buttons do not have a depressed state; (d) descriptions within the manual are mutually inconsistent - e.g., "press the MODE key to switch off AUTO mode" and "in AUTO mode you do not have to use the MODE key, unless you want a different flux mode"; (e) assumptions are omitted - e.g., the manual descriptions only describe changes produced by the buttons and assume that what is unmentioned remains unchanged which is as already stated not what is assumed in MAL. Appendix A provides a set of axioms that combine the results derived from reading the manual with observations from use of the system.

4 Systematic Analysis

Analysis is first concerned with the credibility of the system, exploring those properties that should be true in terms of a plausible mental model of the system. For example:

$$AG(auto \rightarrow on) \qquad (1)$$

The property is described in CTL (Computational Tree Logic, see for example, [4]) and asserts that auto mode can only be armed if the system is on. This property is not true in the version of the system specification based on the manual. A counterexample shows that the air intake key arms the automatic mode without switching the system. A new specification in which the previous state of the system could be recovered even though the system had been switched off fixes the problem. Exploration of other properties indicates that when switching between modes (for example from auto mode to front mode and back) the system keeps a memory of the state in each mode. In the specification a variable *acmem* is used to define the state of the *ac* mode. This and further exploration of system actions produces further changes to the specification (see Appendix A).

The axioms that relate to *acmem* are as follows:

$[ackey]\ acmem' = ac'$
$[a :-\{ackey\}]\ keep(acmem)$
$front \rightarrow [modekey]\ ac' = acmem$
$\neg on \rightarrow [a :\{fanspeedup, fanspeeddown\}]\ ac' = acmem$
$[frontkey]\ \neg front' \rightarrow ac' = acmem$

When *ackey* is pressed, *acmem* stores the new value of *ac* (first axiom), all other actions do not change its value (second axiom – note use of $a{:}-\{ackey\}$ which defines actions *a* not including *ackey*); pressing *modekey* when the front mode is on, puts the air conditioning mode in the state stored in memory (third axiom), and the same happens when *fanspeedup* or *fanspeeddown* are pressed while the system is off (fourth axiom), or if pressing *frontkey* leaves the front mode on (fifth axiom). Property 1 is true in this new model.

Standard patterns were developed for the systematic analysis of interactive systems. Due to space constraints, only minimal information on the patterns is provided, presenting basic (no concurrency) formulations only. The patterns use a number of notational assumptions. $\mathcal{S}$ is the valuation of the attributes in the current state (S), $c \subseteq dom(\rho)$ (with $\rho{:}Attributes \rightarrow Presentation$ defining the presentation modalities) a subset of perceivable attributes, $=_*$ is equality distributed over attributes in the state, *a* an action, $AX_a\, p$ a shorthand for $AX(a \rightarrow p)$ (i.e., in all next states arrived at by a, p holds), $\neq_*$ means at least one attribute must be different, and *pred* an optional predicate used to constrain the analysis to a sub-set of states. The patterns are formulated in a CTL like logic that is transformed into correct CTL by the IVY tool (described in Section 5).

Feedback is a key property of a good user interface that helps the user gain confidence in the effect of actions. It helps create an appropriate mental model of the system. Feedback properties can be verified with the following pattern:

Property Pattern: ***Feedback***
Intent: To verify that a given action provides feedback.
Formulation: $AG(pred(s) \wedge c =_* x \rightarrow AX_a\,(c \neq_* x))$ Under the defined condition (*pred*), the action (*a*) will always cause a change in some perceivable attribute (in *c*).

If the mode key is instantiated in the pattern, i.e., $a \equiv modekey$ and feedback is provided by the airflow indicator (indicator (a) in figure 1), the property can be expressed as:

$$AG(airflow = x \rightarrow AX_{modekey}(airflow \neq x)) \qquad (2)$$

The IVY tool instantiates the pattern, generating five properties, one for each flow mode action. These all hold, suggesting that the airflow indicator provides adequate feedback and therefore mode change is clear. Instantiation of the property with *fans-peedup* and associated indicator *fanspeed* (see indicator (b) in figure 1) produces

$$AG(fanspeed = x \rightarrow AX_{fanspeedup}(fanspeed \neq x)) \qquad (3)$$

The property fails when the fan speed is at maximum (10) and the button does not change speed (or indicator). In practice failure of a property may not be significant. While no other indicator is clear at this limit, this may not be a problem for the user.

Consistency of action is another characteristic of a system that facilitates predictability and learning. Consistency can be internal (between different parts of the system) or external (with other systems). Four buttons which act as on/off switches (A/C, Auto, Mode and Front) look the same and should be internally consistent.

Property Pattern: ***Behavioural consistency***
Intent: To verify that a given action causes consistent effect.
Formulation: $AG(pred(s) \wedge s =_* x \rightarrow AX_{ac}(effect(x,s)))$ with $effect : 2^{(S \times S)}$ characterising the effect the action should have in the state.

This generalization of the Feedback pattern states that the action must always cause the same effect in the user interface. The candidates for test are buttons *ackey*, *front-key*, *airintakekey* and *autokey*, the relevant state is the status of each button (*ac*, *front*, *airintake* and *auto*, respectively), and the desired effect is the toggling of that status. In the case of *ackey*, the pattern gives:

$$AG(ac = x \rightarrow AX_{ackey}(ac = \neg x)) \qquad (4)$$

All the instantiated properties hold when the system is switched on except [*autokey*]. In the case of [*autokey*] the button only turns the mode on, it does not turn it off. One of the interesting features of this design is that when the system is off there are a number of unexpected side effects of pressing some of these buttons that cause changes to subsequent behavior.

Although one form of undo has been analyzed already (for the on/off switches), another relevant pattern is whether there are actions that can undo the effect of other actions.

Property Pattern: ***Undo***
Intent: To check whether the effect of an action can be undone.
Formulation (any action): $AG(s =_* x \rightarrow AX_{a1}EX(s =_* x))$ with a_1□the action whose effect we want to undo, any action required to undo.
Formulation (specific action): $AG(s =_* x \rightarrow AX_{a1}(EX(a_2) \wedge AX_{a2}(s =_* x))$ a_2 the action that should undo a_1; the action availability test (EX(a_2)) is optional.

Property Pattern: ***Reversibility***
Intent: To check whether the effect of an action can be eventually reversed/undone.
Formulation: $AG(s =_* x \rightarrow AX_{a1}EF(s =_* x))$

For the mode button this pattern checks whether there is another action that can be identified as performing its undo. Focussing on the airflow indicator:

$$AG(airflow = x \rightarrow AX_{modekey}AX_{xaction}(airflow = x)) \tag{5}$$

Attempting the verification for $x_{action} = autokey$ fails for all properties, except when $airflow = floorws$. It fails because *modekey* does not have a symmetric action that undoes its effect (on the airflow mode). Exploring why it holds in the one case leads to the unexpected conclusion that the *modekey* action is unavailable when the air flow mode is *floorws*. The mode key action should always be available to allow the flow mode to be changed. The model has been specified so that the user can always press the buttons but this does not imply that pressing a button always has an effect. The problem is that the cyclic behaviour 'implemented' by the mode button includes *wsclear* but this mode should only be accessible by using the Front key. Whether the *modekey* can always be undone by some means leads to a positive answer.

$$AG(airflow = x \rightarrow AX_{modekey}EF(airflow = x)) \tag{6}$$

5 Checking Patterns Using IVY

The IVY tool supports the patterns described in the previous section. Its architecture is given in Figure 2. The tool has four components: a *model editor* designed to support MAL interactor development; a *property editor* designed to support the formulation of relevant usability related properties; a *translator* (i2smv) that transforms interactor models into the model checker's input language; *a trace visualizer/analyzer* that helps analyze any traces produced by the model checker.

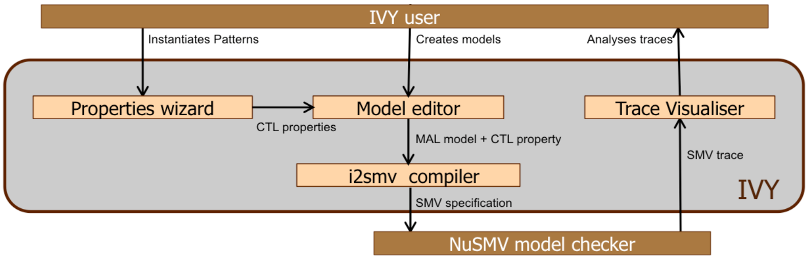
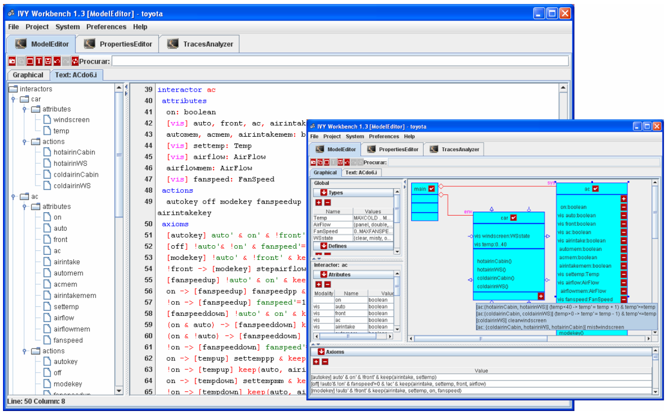


Fig. 2. IVY Architecture

5.1 The Model Editor

The editor supports the structure and syntax of MAL [1] interactors in two editing modes indicated in the two windows of figure 3. In graphical mode the overall structure of the model can be viewed and manipulated while at the same time providing an individual edit capability. The textual mode involves the usual editing facilities: cut and paste, undo and redo etc. This mode supports direct editing and fine tuning. The interactor in graphical mode is based on UML class diagrams [13].

Interactor aggregation and specialization uses an approach consistent with UML to make it easier for designers to understand a model's representation. A number of inspectors are provided in graphical mode to make it possible to edit the different aspects of the model (types, attributes, actions and axioms of the selected interactor, and so on). Textual mode allows direct editing of the text of the model thus enabling experienced users to edit the model more quickly. Aspects of the text can be changed directly instead of using the inspector panels of the graphical mode. Less expert users may choose more guidance through the graphical mode.

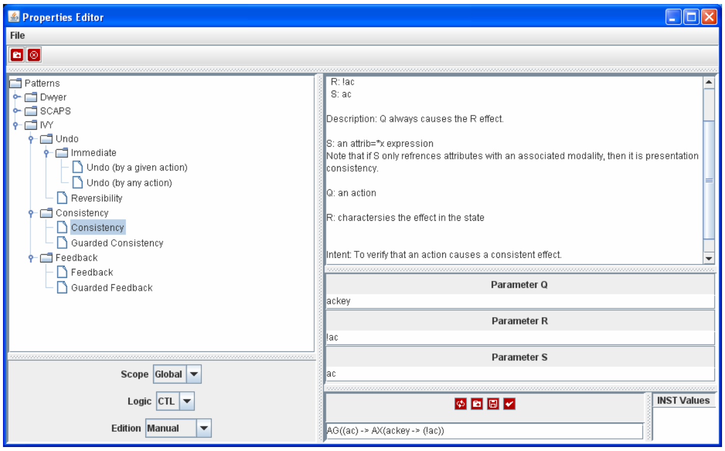


```
interactor ac
 attributes
  on: boolean
  [vis] auto, front, ac, airintak
  automem, acmem, airintakemem: b
  [vis] settemp: Temp
  [vis] airflow: AirFlow
  airflowmem: AirFlow
  [vis] fanspeed: FanSpeed
 actions
  autokey off modekey fanspeedup
airintakekey
 axioms
  [autokey] auto' & on' & !front'
  [off] !auto'& !on' & fanspeed'=
  [modekey] !auto' & !front' & ke
  !front -> [modekey] stepairflow
  [fanspeedup] !auto' & on' & kee
  on -> [fanspeedup] fanspeedpp &
  !on -> [fanspeedup] fanspeed'=1
  [fanspeeddown] !auto' & on' & k
  (on & auto) -> [fanspeeddown] k
  (on & !auto) -> [fanspeeddown]
  !on -> [fanspeeddown] fanspeed'
  on -> [tempup] settemppp & keep
  !on -> [tempup] keep(auto, airi
  on -> [tempdown] settempmm & ke
  !on -> [tempdown] keep(auto, ai
```

Fig. 3. IVY Model Editor

5.2 Property Editor

Verification of assumptions about the expected behavior of the device is achieved by expressing CTL properties. The Property Editor uses patterns to support the choice of specific properties (see figure 4). The editor supports pattern selection, making it easy to instantiate the chosen pattern expressed in CTL (or LTL) with actions and attributes from the model as shown in the figure. Verification is achieved from the translated MAL interactors by the NuSMV model checker [3]. The trace visualizer can then be used to analyze counter-examples or witnesses after the checking process.



R: !ac
S: ac

Description: Q always causes the R effect.

S: an attrib=*x expression
Note that if S only refrences attributes with an associated modality, then it is presentation consistency.

Q: an action

R: charactersies the effect in the state

Intent: To verify that an action causes a consistent effect.

Parameter Q: ackey
Parameter R: !ac
Parameter S: ac

Scope: Global
Logic: CTL
Edition: Manual

AG((ac) -> AX(ackey -> (!ac))

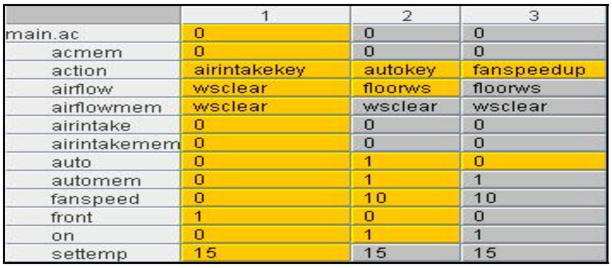
Fig. 4. Expressing properties using patterns

5.3 Trace Visualization

Traces are expressed in terms of the variables and states generated through the translation into SMV's input language. Since the SMV model includes some state artifacts that were created through this step an important element in trace visualization is to ensure that the states and variables that are displayed for the analyst are only in terms of the original interactors. A typical example of this reversion is the elimination of the attribute 'action', annotations used in SMV to distinguish MAL actions. The visualization component aims to focus on the problem that is being pointed out by the trace to support discovery of possible solutions reducing the cost of the analysis.

The visualizer implements a number of alternative representations to explore the acceptability of different approaches. They include: a tabular representation that is similar to the existing SMV implementation of Cadence Labs (www.cadence.com); a graphical representation based on states; and an Activity Diagram representation based on actions [7].

The tabular representation (figure 5) presents information in a table similar to that generated by Cadence SMV or by [12]. Column headings show state numbers. The beginning of a cycle is shown by an asterisk. Cells with darker backgrounds indicate that the attribute's value in the current state has changed since the previous state otherwise a lighter background is used. This idea, adopted from [12], shows quickly when the interactor's attributes change state.



	1	2	3
main.ac	0	0	0
acmem	0	0	0
action	airintakekey	autokey	fanspeedup
airflow	wsclear	floorws	floorws
airflowmem	wsclear	wsclear	wsclear
airintake	0	0	0
airintakemem	0	0	0
auto	0	1	0
automem	0	1	1
fanspeed	0	10	10
front	1	0	0
on	0	1	1
settemp	15	15	15

Fig. 5. Tabular representation: no feedback for *fanspeedup*

The state based representation (see figure 6, left) represents each interactor in a column showing evolution of interactor states (attributes are listed against each state). The global state (including all interactor variables) is represented separately to serve as an index to the states of the individual interactors. A green arrow indicates the beginning and end of loops in this state. Alternatively a pop-up option toggles attribute representation to provide a more compact view (as shown in figure 6). While attributes are not represented in the diagram they can be consulted by placing the mouse over each state, thereby reducing information and making it easier to discover the problem highlighted by the trace. Actions are shown as labels in the arrows between two consecutive states if a transition exists. A second variant of this diagram represents the (physical) states of the SMV modules generated from the model.

The Activity Diagram representation follows the notation of UML 2.0 for activity diagrams (right hand side, figure 6). Activities are represented by one rectangle with rounded corners. The small rectangles associated with the activities represent the state of the interactor before and after an activity occurs. As this representation clearly focuses on actions, interactor attributes appear as pop-ups. The attribute values can be consulted through one pop-up, placing the mouse on the rectangles of the states.

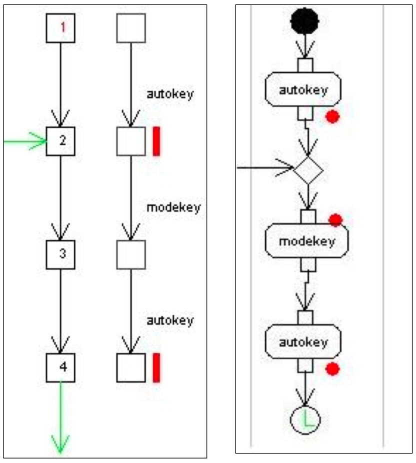


Fig. 6. Counter example representations (state based/activity diagram)

5.4 Exploring the Traces

The visualizer (in all modes) makes it possible to mark states depending on criteria defined over the state attributes. Criteria are defined by relations (=, >, <) between attribute pairs or between attributes and values. To each criterion is associated a color. All the states that verify a given criterion are annotated with the specified color. In the case of figure 6 states marked are states where *airflow = panel.*

In the case of comparison of attributes, two half-circles of the chosen color are drawn near each one of the relevant attributes. In the case of comparison between attributes and values, filled circles are drawn, with the chosen color. If the pop-ups option is enabled the condition represented by each marker can be revealed by placing the mouse over it.

6 Extending the Analysis

Mode complexity is a fundamental issue in interactive system design and is particularly susceptible to model checking analysis. In [8] two types of modes are identified: action modes and indicator modes. Problems might arise when two modes are similar but not the same (leading users to believe the system is in a mode that it is not). Other problems arise through the evolution of modes (for example, actions might cause undesirable/incorrect mode changes) rendering the effect of action unpredictable.

A step beyond the toggling behavior of buttons would be to analyze whether the buttons, when pressed twice, leave the overall mode of the system in the same state. Consider, for example, the front key. If the system is off it always turns the system on. Further investigation could explore a broader concept of "working mode" (a set of state attributes that are related by mode). For example testing whether it is the case that when the system is on, the effect of turning the air flow on and off is to leave the system in the same working mode as it was in initially. For this case the Undo pattern can be used with the specific action formulation, making a_1 and a_2 equal to the *frontkey*. In this case the attributes that are relevant to the working mode include the attributes *auto*, *on*, *ac*, *airintake* and *airflow*. Attributes *settemp*, *fanspeed* and *front* are not relevant to the analysis. Since the action *frontkey* has already been exhaustively analyzed it shall be ignored. Applying the pattern, the following property is produced:

$$AG((auto, on, ac, airintake) =_{*} x \rightarrow \\ AX_{frontkey}(EX(frontkey) \wedge AX_{frontkey}((auto, on, ac, airintake) =_{*} x))))$$

Action modes may be explored using the consistency pattern. When the effect is different from the one expected, action modes can be identified. Alternatively a guard can be used to identify a relevant mode making it possible to check whether the action has the correct behavior for the mode (or, negating the guard and checking whether it has that same behavior outside the relevant mode).

The above analysis limits consideration by ignoring the function of the panel. In the style of [2] an alternative strategy would be to explore how the device enables the environment to reach a desired temperature. This property relates to the context of use of the device, the temperature of the environment, which is not present in the model. There is no space in the paper to present a relevant analysis.

7 Conclusion

For formal techniques to become a widely used approach to the analysis of interactive systems two developments are necessary. The first is to make the analysis commonplace and systematic for developers. The second is to allow reuse of similar specifications to reduce the work necessary to perform the analysis. The work described in this paper addresses both these developments. The use of IVY and patterns provides real promise that systematic techniques are now available for a class of control panel systems. Consideration has been limited to control panel interfaces because the specification of dynamically changing nested actions becomes relatively cumbersome in MAL. The variety and number of such systems that are currently under analysis is growing substantially. The same small set of examples is no longer the focus of attention. Combining tools like IVY with repositories of specifications such as that envisaged by Thimbleby using XML standards (see, for example [9]) will provide an invaluable resource for interactive system developers. The issue of reuse is also being addressed. Patterns provide significant support for developers when they face new designs. Further work is required to explore generic interactors, similar to that discussed in the broader context of smart environments [10].

Acknowledgments. We acknowledge with thanks EPSRC grant EP/F01404X/1 and FCT/FEDER grant POSC/EIA/56646/2004. Michael Harrison is grateful to colleagues in the ReSIST NoE (www.resit-noe.org), José Campos to Nuno Sousa for work in IVY.

References

1. Campos, J.C., Harrison, M.D.: Model checking interactor specifications. Automated Software Engineering 8, 275–310 (2001)
2. Campos, J.C., Harrison, M.D.: Considering context and users in interactive systems analysis. In: van de Veer, G., Palanque, P., Wesson, J. (eds.) Engineering Interactive Systems (accepted for publication, 2007)
3. Cimatti, A., Roveri, M., Olivetti, E., Keighren, G., Pistore, M., Roveri, M., Semprini, S., Tchaltsev, A.: NuSMV 2.3 user manual. Technical report, ITC-IRST, Trento, Italy (2007)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
5. Degani, A.: Taming HAL: designing interfaces beyond 2001. Macmillan, Palgrave (2003)
6. Duke, D.J., Harrison, M.D.: Abstract interaction objects. Computer Graphics Forum 12(3), 25–36 (1993)
7. Fowler, M.: UML Distilled: a brief guide to the standard object modelling language, 3rd edn. Addison-Wesley, Reading (2004)
8. Gow, J., Thimbleby, H., Cairns, P.: Automatic critiques of interface modes. In: Gilroy, S.W., Harrison, M.D. (eds.) DSV-IS 2005. LNCS, vol. 3941, pp. 201–212. Springer, Heidelberg (2006)
9. Gow, J., Thimbleby, H.W.: MAUI: An interface design tool based on matrix algebra. In: Jacob, R.J.K., Limbourg, Q., Vanderdonckt, J. (eds.) Computer Aided Design of User Interfaces IV, CADUI 2004, pp. 81–94 (2004)

10. Harrison, M.D., Kray, C., Campos, J.C.: Exploring an option space to engineer a ubiquitous computing system. Electr. Notes in Theoretical Computer Science 208C, 41–55 (2008)
11. Heymann, M., Degani, A.: Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm. Human Factors: The Journal of the Human Factors and Ergonomics Society 49(2), 311–330 (2007)
12. Loer, K., Harrison, M.D.: An integrated framework for the analysis of dependable interactive systems (IFADIS): its tool support and evaluation. Automated Software Engineering 13(4), 469–496 (2006)
13. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual (UML). Addison-Wesley, Reading (1999)
14. Thimbleby, H.W.: Press on: principles of interaction programming. MIT Press, Cambridge (2007)

Appendix A System Definition

```
defines
 MAXCOLD = 15
 MAXHOT = 30
 MAXFANSPEED = 10
types
 Temp = MAXCOLD .. MAXHOT
 AirFlow = {panel, double, floor, floorws, wsclear}
 FanSpeed = 0..MAXFANSPEED

interactor main
 attributes
  [vis] auto, on, front, ac: boolean
  [vis] airintake: boolean         # true: fresh / false: recirc
  automem, acmem, airintakemem: boolean
  [vis] settemp: Temp
  [vis] airflow: AirFlow
  airflowmem: AirFlow
  [vis] fanspeed: FanSpeed
 actions
  autokey off modekey fanspeedup fanspeeddown tempup tempdown frontkey ackey airintakekey
 axioms
  [autokey] auto’ ∧ on’ ∧ ¬front’ ∧ keep(airintake, settemp)
  [off] ¬auto’ ∧ ¬on’ ∧ fanspeed’=0 ∧ ¬ac’ ∧ keep(airintake,settemp,front,airflow)
  [modekey] ¬auto’ ∧ ¬front’ ∧ keep(airintake,settemp,on,fanspeed)
  ¬front → [modekey] (airflow=panel → airflow’=double) ∧ (airflow=double → airflow’=floor)
           ∧ (airflow=floor → airflow’=floorws) ∧ (airflow=floorws → airflow’=panel) ∧ keep(ac)
  [fanspeedup] ¬auto’ ∧ on’ ∧ keep(airintake, settemp, front, airflow)
  on → [fanspeedup] (fanspeed<MAXFANSPEED → fanspeed’=fanspeed+1)
                    ∧ (fanspeed=MAXFANSPEED → fanspeed’=fanspeed) ∧ keep(ac)
  ¬on → [fanspeedup] fanspeed’=1
  [fanspeeddown] ¬auto’ ∧ on’ ∧ keep(airintake, settemp, front, airflow, ac)
  (on ∧ auto) → [fanspeeddown] keep(fanspeed, ac)
  (on ∧ ¬auto) → [fanspeeddown] (fanspeed >0 → fanspeed’=fanspeed -1)
                                ∧ (fanspeed =0 → fanspeed’=fanspeed) ∧ keep(ac)
  ¬on → [fanspeeddown] fanspeed’=1
  on → [tempup] (settemp<MAXHOT → settemp’=settemp +1)
                ∧ (settemp=MAXHOT → settemp’=settemp) ∧ keep(auto,airintake,on,front,ac)
  ¬on → [tempup] keep(auto,airintake,settemp,on,front,airflow,fanspeed,ac)
  on → [tempdown] (settemp>MAXCOLD → settemp’=settemp -1)
                  ∧ (settemp=MAXCOLD → settemp’=settemp) ∧ keep(auto,airintake,on,front,ac)
  ¬on → [tempdown] keep(auto,airintake,settemp,on,front,airflow,fanspeed,ac)
  on → [frontkey] on’ ∧ front’=¬front ∧ keep(settemp)
  ¬on → [frontkey] on’ ∧ front’ ∧ keep(settemp)
  [frontkey] front’ → (¬auto’ ∧ ¬airintake’ ∧ ac’)
  front ↔ airflow=wsclear
  on → [ackey] ac’=¬ac ∧ keep(auto,airintake,settemp,on,front,airflow,fanspeed)
  ¬on → [ackey] keep(auto,airintake,settemp,on,front,airflow,fanspeed,ac)
  [airintakekey] airintake’=¬airintake ∧ keep(auto,settemp,on,front,airflow,fanspeed,ac)
  [ ] ¬auto ∧ ¬on ∧ fanspeed=0 ∧ ¬ac
 # airflow
  ¬front → [frontkey] airflowmem’=airflow
  front → [ac:-{frontkey, modekey}] keep(airflowmem)
  front → [modekey] airflow’=airflowmem
  (on ∧ front) → [frontkey] airflow’=airflowmem
  (¬on ∧ front) → [frontkey] keep(airflowmem)
 # airintake
  ¬front → [frontkey] airintakemem’=airintake
  front → [ac:-{ffrontkey, airintakekeyg}] keep(airintakemem)
  front → [airintakekey] airintakemem’=airintake’
  (on ∧ front) → [frontkey] airintake’=airintakemem
  (¬on ∧ front) → [frontkey] keep(airintakemem)
 # ac
  [ackey] acmem’=ac’
  [ac:-{ackey}] keep(acmem)
  (front ∧ on) → [modekey] ac’=acmem
  (front ∧ ¬on) → [modekey] keep(ac)
  ¬on → [ac:{fanspeedup,fanspeeddown}] ac’=acmem
  [frontkey] ¬front’ → ac’=acmem
  [autokey] ac’=acmem
 # auto
  [ac:{autokey,modekey}] automem’=auto’
  [ac:-{autokey,modekey,frontkey}] keep(automem)
  ¬on → [frontkey] keep(automem)
  on → [frontkey] automem’=auto
  [frontkey] ¬front’ → auto’=automem
```