Chapter 1

MAUI: AN INTERFACE DESIGN TOOL BASED ON MATRIX ALGEBRA

Jeremy Gow and Harold Thimbleby
{j.gow, h.thimbleby}@ucl.ac.uk
UCL Interaction Centre (UCLIC), University College London, Remax House, 31–32 Alfred Place, London WC1E 7DP, UK

Abstract We describe MAUI, a user interface design tool that is based on a matrix algebra model of interaction. MAUI can be used to build and analyse designs for interactive systems, such as handheld devices. This paper describes MAUI, its advantages and underlying mathematical approach. MAUI is implemented in Java and XML, which allows flexible integration with other parts of the design life cycle, such as prototyping, implementation and documentation.

Keywords: User interface design, matrix algebra, finite state machines, XML

1. Introduction

Regardless of how attractive they are, many interactive systems remain complex and hard to use, and many result in frustration and accidents. They are often built informally, and it is not obvious what their problems are nor how to avoid them. The research field of HCI aims to improve the user experience, but it suffers from a lack of analytic tools that both support clear formal reasoning and support design and evaluation at a practical scale. The theoretical approaches that have the formal power to specify interactive systems are technical and beyond the reach of real designers; and the practical development tools that create real interactive systems are so informal that systems are inevitably developed in *ad hoc* ways.

This paper introduces MAUI, a matrix algebra based user interface development and analysis tool that provides a simple, general and rigorous approach to design. It is sufficiently powerful to handle many

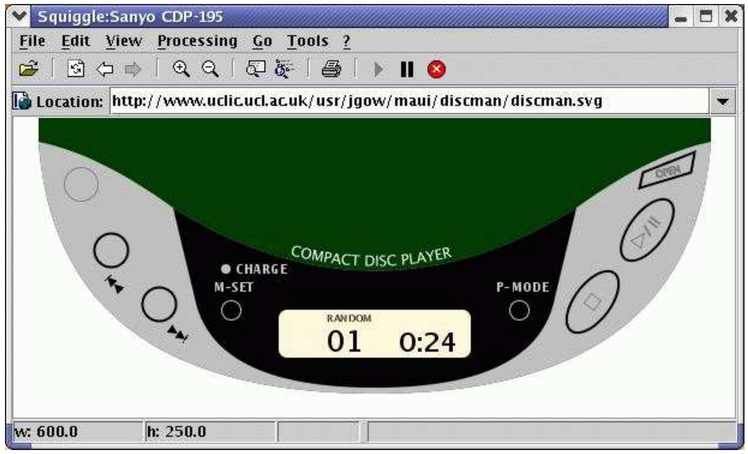

Figure 1.1. An SVG simulation of the Sanyo CDP-195 portable CD player. The graphics are hand-coded, but the simulation code is automatically generated from the interface design in MAUI. Viewed using the Squiggle SVG browser.

complex interactive devices and because of its simplicity raises clear and well-defined design and research questions.

MAUI allows the designer to model an interactive device as a finite state machine (FSM), a technique that has successfully been used in HCI (Thimbleby, 1993). From this representation, an *event algebra* is generated, essentially a decomposition of the FSM's transition matrix into matrices representing individual user actions (Thimbleby, in press). We represent the FSM in linear algebra to permit equational reasoning about user interface events. Properties of the interface can be formally stated as theorems of this event algebra, and checked efficiently via matrix calculations — though a user of MAUI need not know or care about the internal implementation technique. MAUI stands for **M**atrix **A**nalysis of **U**ser **I**nterfaces.

There are three key ideas behind the system:

Specification Algebraic properties can correspond to usability issues. This is explored in Sections 1.5 and 1.6. MAUI maintains an algebraic specification which can be checked against the evolving design.

Simplicity The simplicity of the formalism means that the system can verify and generate relevant properties automatically. Hence the designer does not need to get involved in proof, and can gain insights into the interface design from properties and inconsistencies pointed out by MAUI.

Integration MAUI allows integration with other design tools and processes via XML. For example, fast prototyping with SVG (Scalable Vector Graphics) (Ferraiolo et al., 2003), an XML open standard version of Flash.

Figure 1.1 shows a user interface simulation in SVG. The interface design was specified in MAUI, and automatically combined with an SVG image to make an interactive graphical simulation.

2. FSM Models

Formal techniques have found a wide variety of applications in user interface design — e.g., for a collection of recent work see (Palanque and Paterno, 1997). Finite state machines are a basic formalism with a long history in this area, starting with Parnas (Parnas, 1964) and Newman (Newman, 1969) in the 1960s, and reaching a height of interest in user interface management systems (UIMS) work (Wasserman, 1985); see (A. Dix and Beale, 1998) for a textbook introduction with applications of FSMs in HCI.

Finite state machines (FSMs) are a simple and well understood formalism used throughout computer science. An FSM consists of a finite set of states connected by labelled transitions. In this paper we assume that the states are those of the user interface, and that labelled transitions correspond to those events that change the interface's state. Events usually consist of user actions, but may include other influences on the system. Examples of events are the user pressing a button, selecting a menu item or doing nothing for two seconds. We denote events with a box notation: `Event`.

Figure 1.2 shows an extremely simple example: an FSM model of a light switch. It has the states On and Off, and a `Switch` event that flips between them. This model is *deterministic*, in that every event has at most one effect in any state. A non-deterministic version might define `Switch` in the Off state so that it may turn the light on or blow the bulb. The model in Figure 1.2 is also *unguarded*, in that every event is possible in every state. A guarded version might have a light switch that can be flicked `Up` or `Down` (together replacing `Switch`), where `Up` works only in the On state and `Down` only in the Off state.

Formally, an FSM is a tuple $\langle S, \Sigma, s_0, \delta\rangle$, where S is a set of states, Σ an alphabet (of events, in this case), $\mathbf{s}_0 \in S$ the initial state, $\delta \subseteq S \times \Sigma \times S$ the transition relation. The definition is standard. In MAUI, however, the FSM model is enhanced in two ways: with *signs* and *state classes*.

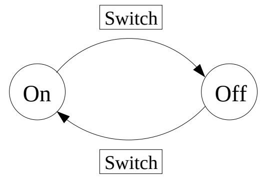

Figure 1.2. A simple FSM model of a light switch.

Signs allow the designer to distinguish between the interface's state and those features observable by the user. An interface has a collection of signs, and each state displays some subset of them. Examples of signs are highlighting a menu item, displaying the time, or playing some music. Each sign may be associated with several states. Formally, we add to the FSM tuple a set of signs Ψ and a function $\omega\colon S \to \mathbb{P}(\Psi)$, which yields the subset of observable signs in each state.

State classes are used to reduce the effort in describing interface models, and for MAUI to classify theorems. Event transitions and signs only have to be defined once for a state class, and are inherited by all the states that are members of the class. A state may be a member of several state classes. Two classes are allowed to assign different transitions to the same state and event — the model will simply be non-deterministic. State classes are presentational and do not change the semantics.

As a modelling technique, FSMs have the advantage of being a standard, simple formalism, and therefore more accessible to the technically-minded interface designer. They are also easy to simulate, which is is good for prototyping.

FSMs can be used *in theory* to model any finite, discrete concurrent or sequential system, and so are widely applicable to user interface design. For example, a related state diagram formalism is used in (Jacob et al., 1999) to model virtual environments.

However, FSMs also have a well known disadvantage in that they scale badly. Because each state is represented explicitly, the size of an FSM increases dramatically with the complexity of the modelled system — a combinatorial explosion. This is a potential problem, as the model may become too large for the designer to comprehend or for a computer to store and analyse.

Fortunately, there are a number of ways in which the combinatorial explosion can be mitigated:

Abstraction Details of the design can be excluded from the model. Useful formal analyses can be still be carried out on abstract models.

Modularisation Large interface designs can often be broken down into a number of distinct, independent models.

Higher-Level Formalisms Models can be built in equivalent higher-level formalisms and compiled down to FSMs for analysis. The designer need never see the underlying FSM; this is the approach of Esterel (Berry, 1998), LTSA (Magee, 1999) and other languages.

Implementation techniques There are numerous compact implementation techniques appropriate for FSMs, including BDDs (Drechsler, 1998) and symbolic techniques.

Pragmatism MAUI works with an event algebra that captures user interface properties; if there is an unmanageable combinatorial explosion then this *might* suggest that the user model is also extremely complex. Thus we claim that if MAUI cannot handle the specification of the device, the designer should have a good idea of *why* the FSM is so complex, how the users will cope with it, and whether this is acceptable.

3. Event Algebras

Analysis in MAUI uses a formalism consisting of states and events, represented by vectors and matrices respectively. For example, the states On and Off from Figure 1.2 are represented as vectors:

$$\mathbf{s}_{\mathit{off}} = (1 \;\; 0) \qquad \mathbf{s}_{\mathit{on}} = (0 \;\; 1)$$

Events are represented as matrices that transform these state vectors according to the FSM model. For example:

$$\boxed{\texttt{Switch}} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Checking that these definitions conform to Figure 1.2 is a matter of elementary matrix multiplication:

$$\mathbf{s}_{\mathit{off}}\,\boxed{\texttt{Switch}} = \mathbf{s}_{\mathit{on}} \qquad \mathbf{s}_{\mathit{on}}\,\boxed{\texttt{Switch}} = \mathbf{s}_{\mathit{off}}$$

This can be read purely algebraically as a description of the light switch, without reference to the underlying vectors and matrices. However, the real advantage is that these matrices form an *event algebra* in which we can make assertions about user actions *independently of any particular state* (Thimbleby, in press). For our toy example, we can state the following property:

$$\boxed{\texttt{Switch}}\,\boxed{\texttt{Switch}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

where I is the identity ("do nothing") matrix. This tells us that pressing $\boxed{\texttt{Switch}}$ twice has the same final effect as doing nothing! This is an inherent property of $\boxed{\texttt{Switch}}$, no matter what state the system is in.

The strength of this approach is that similarly concise statements can be made about far more complex interfaces with many states. We look at some more interesting examples below.

Given a MAUI interface model $\langle S, \Sigma, s_0, \delta, \Psi, \omega \rangle$ we formally define its event algebra with a bijection $\eta\colon \{1 \ldots |S|\} \leftrightarrow S$ mapping states to element indices; η generates a representation function $\mathcal{R}$ that maps states and events to the vectors and matrices that denote them. For state $s \in S$ define the *state vector* $\mathbf{s} = \mathcal{R}[\![s]\!]$ by

$$\mathbf{s}_i = \begin{cases} 1 & \text{if } \eta(i) = s \\ 0 & \text{otherwise} \end{cases}$$

For event $\boxed{\texttt{E}} \in \Sigma$ define the *event matrix* $E = \mathcal{R}[\![\boxed{\texttt{E}}]\!]$ by

$$E_{ij} = \begin{cases} 1 & \text{if } \delta(\eta(i), \sigma, \eta(j)) \\ 0 & \text{otherwise} \end{cases}$$

The algebra of these vectors and matrices, equipped with multiplication and an initial state vector $\mathcal{R}[\![s_0]\!]$, provides another model of the user interface, based on the original FSM. For brevity in this paper, we write the event $\boxed{\texttt{E}}$ to denote the matrix $E = \mathcal{R}[\![\boxed{\texttt{E}}]\!]$; in general capital letters $A, B \ldots$ denote matrices that may or may not be events or products of events.

4. Using MAUI

MAUI's own interface, Figure 1.3, is a conventional GUI design, with windows representing different aspects of a system's functionality: *Design*, *Simulation*, *Statistics* and *Analysis* (described in Section 1.5). There are also menus for basic functions such as opening and saving files.

The Design window displays the current interface design and allows the user to edit it. The window is split into a *Components* panel and a *Relationships* panel. The Components panel can be set to display a list of either states, state classes, events or signs. Selecting a component from this list results in the *Relationships* panel displaying a list of related components. The type of related components displayed can be set by the user.

For example, selecting a state from the Components panel causes its state transitions to be displayed in the Relationships panel. The user can also choose to view the classes the state belongs to, or the signs

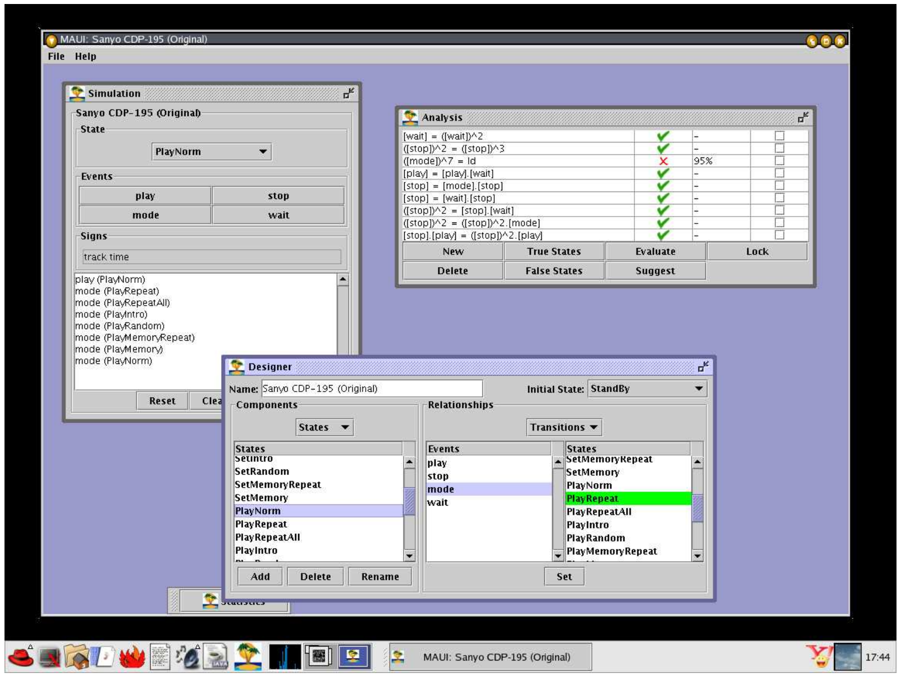

Figure 1.3. MAUI being used to analyse the design of the Sanyo CDP-195 from Figure 1.1.

associated with it. The Components lists may have items added, deleted or renamed, and the Relationships panel may be used to edit transitions, class membership, etc.

The Simulation window shows an interactive simulator, ideal for basic tests. The Simulation window does not aspire to be photorealistic, which is currently handled externally by SVG and other mechanisms.

The Statistics window shows statistics that are useful for comparing the complexity of different designs. For example, minimum, maximum and average path length between two states (Thimbleby, 1993). Another example is the overshoot recovery cost. A common user error is an overshoot caused by doing an event, say $\boxed{\text{E}}$, once too often. MAUI can calculate the overshoot recovery cost as the minimum number of events that correct an overshoot: it determines a product of events R such that $\boxed{\text{E}}\boxed{\text{E}}R = \boxed{\text{E}}$.

The Analysis window allows the user to explore the interface design's event algebra, as discussed in Section 1.3. The following section describes how this works in MAUI, and its utility in user interface design.

5. User Interface Analysis in MAUI

Event algebras in themselves are simply a restatement of an FSM with the transition function 'broken up' into individual events. This makes them well-suited for making statements about how events interact with each other, and hence for usability analysis. Crucially, matrices allow theorems to be checked efficiently by elementary numerical calculation.

Of course, reflecting on the usability of an interface design is an extremely context-dependent process. A formal approach does not relieve the designer of the need to think about the implications of their design, and decide which formal properties are relevant to the user's experience. What event algebras provide is a well-defined language to talk about user interfaces concisely.

MAUI allows the designer to specify a set of event algebra properties that they wish their design to conform to. As the design evolves, the system provides feedback on which parts of the specification are currently satisfied.

Consider an interface button $\fbox{\texttt{A}}$ such that $\fbox{\texttt{A}}\ \fbox{\texttt{A}} = \fbox{\texttt{A}}$, an idempotence that tells us that if $\fbox{\texttt{A}}$ needs pressing, it only ever need be pressed once. The button would avoid the possibility of an overshoot error (pressing once too often). This would be suitable for the specification of a $\fbox{\texttt{Play}}$ or $\fbox{\texttt{Stop}}$ button.

Another example is undo. Allowing the user to undo their actions is a common usability requirement. We can express the requirement that user actions $\fbox{\texttt{B}}\ldots\fbox{\texttt{C}}$ act an an undo for action $\fbox{\texttt{A}}$ by:

$$\fbox{\texttt{A}}\ \fbox{\texttt{B}}\ldots\fbox{\texttt{C}} = I$$

The designer may want each event to be easily undone, and so have a short undo sequence (ideally one action) for each event $\fbox{\texttt{A}}$. Some events are inherently irreversible, and so have no $\fbox{\texttt{B}}\ldots\fbox{\texttt{C}}$ that yields the identity. This can be determined by straightforward calculation (to show the matrix is singular); the designer can specify in MAUI that an event must be reversible, or that it must be irreversible. Further, some events although in principle invertible, are merely irreversible for the user, as there is no sequence of events whose corresponding matrix product is the inverse of the event.

Another kind of usability issue the designer may be interested in is permissivenes (Thimbleby, 2001): allowing many different sequences of actions to achieve any given task, ones that commute or distribute, etc:

$$\begin{aligned}
\fbox{\texttt{A}}\ \fbox{\texttt{B}} &= \fbox{\texttt{B}}\ \fbox{\texttt{A}} \\
\fbox{\texttt{A}}\ \fbox{\texttt{B}}\ \fbox{\texttt{C}} &= \fbox{\texttt{A}}\ \fbox{\texttt{B}}\ \fbox{\texttt{A}}\ \fbox{\texttt{C}} \\
\fbox{\texttt{A}}\ \fbox{\texttt{B}} &= \fbox{\texttt{C}}\ \fbox{\texttt{D}}\ \fbox{\texttt{E}}
\end{aligned}$$

A related usability concept is that efficient shortcuts should be available for expert users: $\boxed{\mathtt{A}}\ \boxed{\mathtt{B}} \ldots \boxed{\mathtt{C}}\ \boxed{\mathtt{D}} = M$ — where, in turn, M can factored as a product of user events, but its total cost (to the user) is less.

So far we have shown how universal statements about interface models can be made in MAUI. In some cases a property will only be of interest for a certain subset of states. This can be done by restricting properties to particular state classes. For example, we can claim for a class, 'For C: $\boxed{\mathtt{A}} = \boxed{\mathtt{B}}$' if for all $\mathbf{s}_{\eta(i)} \in C$ the ith row of $\boxed{\mathtt{A}}$ and $\boxed{\mathtt{B}}$ are equal. Another use for state classes is dealing with predictable effects of actions. We can state that event $\boxed{\mathtt{A}}$ always puts an interface into one of the states in class C if we can show that for every non-zero jth column of $\boxed{\mathtt{A}}$ the state $\mathbf{s}_{\eta(j)}$ is not in C.

The designer manages the specification through MAUI's Analysis window. This presents a list of the currently specified properties, with options to add, delete and edit them. MAUI distinguishes between three basic types of property: equality of two event/state expressions; the reversibility of an individual event; and predictability of an event. Choosing to create or edit a property brings up an editing panel that allows these properties to be composed from the existing events, states and state classes in a straightforward way. Predefined events and states, like the identity and so on, are also provided. More complicated properties can be built up in the editing panel by either negating properties or restricting properties to a particular state class.

The Analysis window monitors how the current interface design conforms to the designer's specification. Unsatisfied properties are highlighted, and annotated with a percentage of *how* true they are. For an equality theorem the percentage of states for which it holds is used, by calculating the percentage of equal matrix rows. Other measures could be used. The designer can also request detailed information about why a property is not true in the form of counter-example states, and can 'lock' any true property, so that MAUI forbids changes to the user interface that make it false.

One feature that makes MAUI stand out as a design tool is its ability to suggest to the designer properties of the interface model. At the designer's request the system can automatically generate true theorems not already in the specification, as well as 'near-theorems' — non-theorems of the equality type that are true for a high percentage (e.g., $> 95\%$) of states. The value of near-theorems is that they may represent properties which the designer could choose to make universal, for a more clear and consistent design.

The automatic suggestion mechanism currently works by enumerating all identities up to a certain complexity, with some redundant theorems being pruned because they can be dervied from simpler theorems. In order to manage the amount of suggestions generated by MAUI, the designer can vary both the theorem complexity level and the percentage threshold for near-theorems.

Ivory and Hearst (2001) point out the current lack of automated support for critiquing user interfaces, that is "methods that not only point out difficulties but propose improvements." Following their terminology, MAUI's ability to suggest properties that are *or should* be true is a simple form of support for critiquing analytical models. No other technique in their survey provides this kind of support.

6. Examples

The MAUI suggestion mechanism was used to analyse the design of a portable CD player, the Sanyo CDP-195. The 29 state model captured the behaviour of four events: $\boxed{\texttt{Play}}$, $\boxed{\texttt{Stop}}$, $\boxed{\texttt{Mode}}$ and $\boxed{\texttt{Wait}}$ (for 6 seconds). The $\boxed{\texttt{Mode}}$ button selects one of seven play modes (Normal, Random, Intros, ...). The suggestion mechanism generated the following 95% near-theorem:

$$\boxed{\texttt{Mode}}^7 = I \tag{1.1}$$

Reflecting on why this is almost universally true, we found that the $\boxed{\texttt{Mode}}$ button cycled through the seven play modes and returned to the original state, irrespective of whether the player was at rest, playing or paused — *except for in one state*. In this state the display gave the CD track/time information, but $\boxed{\texttt{Mode}}^7$ took the user to an equivalent state with no display except '--'. Merging these two states would have no effect on the functionality of the interface, but would make (1.1) true and, we suggest, the device more understandable to the user. MAUI's suggestion for a design property thus leads to a simpler and more consistent interface design.

As a second example, the Nokia 5510 mobile phone menu system (Thimbleby, in press) can be specified by 5 event matrices, over 188 states. We can automatically (and quickly) find theorems including:

$$\begin{aligned}
\boxed{\texttt{Up}}\boxed{\texttt{Cancel}} &= \boxed{\texttt{Cancel}} \\
\boxed{\texttt{Down}}\boxed{\texttt{Cancel}} &= \boxed{\texttt{Cancel}} \\
\boxed{\texttt{Cancel}}^4 &= \boxed{\texttt{Cancel}}^5 \\
\boxed{\texttt{Up}}\boxed{\texttt{Down}} &= I
\end{aligned}$$

7. Design Integration via XML

MAUI can store user interface designs in an XML format. This is ideal for integrating the formal analysis done in MAUI with other stages of the design cycle: prototyping, documentation, implementation, alternative analysis tools etc. For proof-of-concept, so far we have written XSLT stylesheets to convert designs to:

Graphviz Visualisations of interface state graphs were produced by converting XML designs into AT&T's Graphviz format (Gansner and North, 2000).

HTML+Javascript HTML simulations are a simple, portable way to share designs with other people over the web.

SVG+Javascript Hand-coded SVG (Ferraiolo et al., 2003) was added to the MAUI-generated XML and automatically transformed into SVG+Javascript, for a more sophisticated graphical simulation. We intend to adapt an existing SVG editor to integrate a graphical design editor with MAUI, to avoid the need to write the SVG graphical elements by hand, as at present.

Mathematica In the hands of an expert user, *Mathematica* could do larger and far more complex analyses than are done in MAUI, although it is far less accessible than our system, both in terms of ease of use and price (MAUI is free).

Reusing the design data in each stage means there is no need to reimplement the design several times, with the possibility of errors occuring at each stage. Figure 1.4 shows fragments of XML describing the Sanyo CDP-195 mentioned in Section 1.5. The XML was generated by MAUI, except for the hand-coded `form` element which contains the graphical design. It was automatically transformed to the graphical simulation shown in Figure 1.1.

8. Further Work

In developing MAUI our highest priority is to apply it to more real-world case studies. We have argued for the generality of MAUI's design methodology, and given some examples. However, further work with a wider range of examples is needed to establish the scope of the method, both in terms of types of system and types of usability analysis.

MAUI is a research tool, but a separate question is how accessible we could make our formal methodology to designers or HCI researchers. The real questions here is 'which ones?' MAUI's approach to formal

```
<fui>
   <name>Sanyo CDP-195</name>
   <event id="play"/>
   <event id="mode"/>
   ...
   <form width="600" height="250">
     ...
     <signs>
        <text id="track" ... x="260" y="195">01</text>
        <text id="time" ... x="320" y="195">0:24</text>
        ...
     </signs>
   </form>
   <function>
     <initial ref="StandBy"/>
     <state id="StandBy">
        <change event="play" to="PlayNorm" />
     </state>
     <stateclass id="PlayState">
        <change event="stop" to="NoAction" />
     </stateclass>
     <state id="PlayNorm" class="PlayState">
        <change event="play" to="PauseNorm" />
        <change event="mode" to="PlayRepeat" />
        <sign ref="track"/>
        <sign ref="time"/>
      </state>
      ...
   </function>
</fui>
```

Figure 1.4. XML description of the Sanyo CDP-195 portable CD player generated by MAUI, except for the content of the `form` element, which is hand-coded SVG.

analysis is an attempt to be simple enough for more technically-minded designers to grasp and to still be useful. Any further development will need to consider more about the abilities and requirements of designers and/or HCI researchers.

Sometimes a user will follow a detour to achieve some straightforward goal, as in $AB \ldots CD = AD$, etc. An interesting future development might be to make some of MAUI's analyses available to end users, not just designers. "Would you like to know a better way to do what you have just done?" In Hyperdoc (Thimbleby, 1993), the end user could ask the system to find event sequences that set signs to particular values.

There are many techniques for compressing matrices. In MAUI, an interesting possibility to explore would be to compress matrices and hence help a designer determine tighter class definitions and nearly (or completely) redundant transitions, as well as transitions that if changed might reduce the model.

MAUI's statistics could be extended in many ways, such as incorporating expectations based on Markov models (Thimbleby et al., 2001). MAUI could constrain design changes to maintain statistics, as it currently does for theorems.

9. Conclusions

We have described MAUI, a design tool in which formal models of user interfaces can be built and analysed. Integration with other design processes, especially graphical prototyping, is achieved using XML. Design specifications are expressed and easily verified using event algebras, with the novel feature that the system can suggest to the designer properties that are true or nearly true.

Our approach can be related to a great deal of previous work on modelling user interfaces with finite state machines and related formalisms. For instance, VEG (Berstel et al., 2001) is a recent example based on BNF grammars. MAUI's algebraic style of specification, based on the global properties of events, is a key difference with such methods. Also, more sophisticated interface models are typically employed in order to ease the specification process. This is a less important difference, as such techniques could be adopted by MAUI.

Many systems, like LTSA (Magee, 1999) or the Play-Engine (Harell and Marelly, 2003), aim for comprehensiveness, and thus tend to lose sight of clarity in usability and effective use by typical mathematically naïve designers. Usability itself is a very complex field, and we feel that the interaction between usability research and various schemes for combining rapid prototyping and modelling are not best helped by the usual goals of universality.

We imagine that as a body of design and usability related theorems is developed (e.g., that many pairs of actions, such as `Up` and `Down`, should be inverses), these will be embedded into MAUI, thus making it a convenient tool for designers and researchers not only to build, simulate and generate prototype interactive systems, but to check a wide range of their properties.

Acknowledgements Harold Thimbleby is a Royal Society Wolfson Research Merit Award Holder. Jeremy Gow is funded on the award. We are grateful to Paul Cairns for constructive comments.

References

A. Dix, J. Finlay, G. A. and Beale, R. (1998). *Human Computer Interaction*. Prentice Hall, 2nd edition.

Berry, G. (1998). The foundations of Esterel. In Plotkin, G., Stirling, C., and Tofte, M., editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press.

Berstel, J., Reghizzi, S. C., Roussel, G., and Pietro, P. S. (2001). A scalable formal method for design and automatic checking of user interfaces. In *Proc. 23rd International Conference on Software Engineering (ICSE'01)*, pages 453–462. IEEE Computer Society.

Drechsler, R. (1998). *Binary Decision Diagrams: Theory and Implementation*. Kluwer.

Ferraiolo, J., Jackson, D., and Jun, F. (2003). Scalable Vector Graphics (SVG) 1.1 Specification. Recommendation, W3C. Available online at `http://www.w3.org/TR/SVG11`.

Gansner, E. and North, S. (2000). An open graph visualization system and its applications to software engineering. *Software Practice & Experience*, 30(11):1203–1233.

Harell, D. and Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer Verlag.

Ivory, M. Y. and Hearst, M. A. (2001). The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4):470–516.

Jacob, R., Deligiannidis, L., and Morrison, S. (1999). A software model and specification language for non-wimp user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46.

Magee, J. (1999). Behavioral analysis of software architectures using LTSA. In *Proc. 21st International Conference on Software Engineering (ICSE'99)*, pages 634–637. ACM.

Newman, W. M. (1969). A system for interactive graphical programming. In *Proc. 1968 Spring Joint Computer Conference*, pages 47–54. American Federation of Information Processing Societies.

Palanque, P. and Paterno, F. (1997). *Formal Methods in Human-Computer Interaction*. Springer Verlag.

Parnas, D. L. (1964). On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proc. 24th. ACM National Conference*, pages 379–385.

Thimbleby, H. (1993). Combining systems and manuals. In Alty, J. L., Diaper, D., and Guest, S. P., editors, *People and Computers VIII, HCI'93*, pages 479–488. Cambridge University Press.

Thimbleby, H. (2001). Permissive user interfaces. *International Journal of Human Computer Studies*, 54(3):333–350.

Thimbleby, H. (in press). User interface design with matrix algebra. Accepted by *ACM Transactions on Computer Human Interaction*.

Thimbleby, H., Cairns, P., and Jones, M. (2001). Usability analysis with markov models. *ACM Transactions on Computer-Human Interaction*, 8(2):99–132.

Wasserman, A. I. (1985). Extending state transition diagrams for the specification of human computer interaction. *IEEE Transactions on Software Engineering*, SE-11(8):699–713.