Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 183 (2007) 57–72
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Formal Models for Informal GUI Designs

Judy Bowen[1,2] and Steve Reeves[3]

Department of Computer Science
University of Waikato
Hamilton, New Zealand

Abstract

Many different methods exist for the design and implementation of software systems. These methods may be fully formal, such as the use of formal specification languages and refinement processes, or they may be totally informal, such as jotting design ideas down on paper prior to coding, or they may be somewhere in between these two extremes. Formal methods are naturally suited to underlying system behaviour while user-centred approaches to user interface design fit comfortably with more informal approaches. The challenge is to find ways of integrating user-centred design methods with formal methods so that the benefits of both are fully realised. This paper presents a way of capturing the intentions behind informal design artefacts within a formal environment and then shows several applications of this approach.

Keywords: Formal methods, user-centred design, GUIs, refinement, informal design artefacts.

1 Introduction

When we are designing and building systems, particularly large and complex systems, it is not unusual to work in a modular fashion. Different parts of the system will be worked on at different times, perhaps by different groups of software engineers, designers and programmers.

Separation of the design and implementation of a graphical user interface (GUI) of a system from what we will refer to as the underlying system behaviour is a common and pragmatic approach for many applications. The development of user interface management systems (UIMS) based on the logical separation of system functionality and user interface (UI) is exemplified by the Seeheim model [20]. The separation allows us to not only focus on the different concerns which different parts of the system development present, but, more importantly, allows for different approaches and design techniques.

[1] Thanks to the anonymous reviewers for DSV-IS 2006

[2] Email: jab34@cs.waikato.ac.nz

[3] Email: stever@cs.waikato.ac.nz

doi:10.1016/j.entcs.2007.01.061

When we develop the underlying system functionality for an application we are often concerned with issues such as correctness, reliability, robustness and efficiency *etc.* which lend themselves to the techniques we call "formal". Such formal techniques include specifying requirements, validating and verifying specifications and refinement methods. When we develop UIs, however, our concerns are often more human-focussed (this is particularly true if we follow a user-centred design (UCD) approach). The design techniques we adopt reflect this and rely on more informal strategies such as prototyping, scenarios and storyboards, iteration based on user-feedback, usability testing *etc.*

Whilst we can see the benefits of this separation of concerns and design methods in terms of being able to adopt the most suitable development approach to different parts of the task, there are clearly some problems associated with it. If our aim is to use a formal process to develop provably correct software (which it is), then we must ensure that all parts of the system have been designed in a way which satisfies this.

This gap between the formal and informal has been identified and discussed many times, notably in 1990 by Thimbleby [24]. Several different approaches have been taken over recent years by different groups of researchers to try and bridge this gap. Much of the work that has been done falls into one of the following categories:

- Development of new formal methods for UI design. *E.g.* Modelling UIs using new formalisms [7];
- Development of hybrid methods from existing formal methods and/or informal design methods. *E.g.* Using temporal logic in conjunction with interactors [17];
- Use of existing formal methods to describe UIs and UI behaviour. *E.g.* Matrix algebra for UI design [25];
- Replacing existing human-centred techniques with formal model-based methods. *E.g.* Using UI descriptions in Object-Z [23] to assess usability [12].

Whilst much of this work is demonstrably a step forward in bringing together formal methods and UI design, the methods and techniques which have been developed have failed, in the most part, to become mainstream.

One of the reasons for this seeming reluctance for either group to adopt the new methods proposed is, of course, the reluctance of any group to change working practices which are meeting their individual needs. Persuading users of formal methods to adopt less formal, or new hybrid, methods has proved as unsuccessful as encouraging UI designers to abandon their human-centred approach in favour of more formal approaches.

Rather than trying to change the methods used by different groups of software developers, the approach we are taking with our research is to consider the existing, diverse, methods being used and develop ways of formally linking them together. In particular, because our interests lie in both using formal methods and rigorous development techniques to develop our software, and UCD approaches to UI design, our intention is to find ways of interpreting the sorts of informal design artefacts produced in a UCD process within our formal framework.

In this paper we will introduce a way of formally describing informal design artefacts, called the *presentation model*. We will give some examples of the use of the presentation model within a formal design context and then show how we can extend this model with another formalism, finite state machines. We can then begin to explore both the static and dynamic meanings of the designs which form the basis of the model.

2 User-Centred Design Artefacts

The purpose of user-centred design is to ensure that the software we build, and in particular the interface to that software, meets the expectations of the intended users. To this end the processes used are designed to involve users from an early stage to find out about not only the tasks they need to perform with the software, but also things like the current working practices of the users, their experience with similar software, internal company working processes that will be affected by this new software, *etc.*

Techniques used early in a UCD process may include ethnographic studies which allow the designers to understand not only the users, but also their work environment and work processes. This may be followed by task analysis methods to examine the users' requirements of the system. Task analysis has received a lot of attention from formal practitioners over the years, and a number of models exist for this, as well as methods for developing UIs from such models, *e.g.* [6], [18]. UCD practitioners may use scenarios and personas to enhance the task analysis process and give details of specialised requirements and user behaviours.

The actual design of the UI may involve brainstorming sessions between designers and users which will lead to the development of prototypes. These prototypes are then tested by both users and design specialists and updated in an iterative process before a final design is reached. Even this final design is subject to amendment once the system has been implemented and subsequently undergone usability testing.

The key to UCD, therefore, is to ensure that the actual users of the system are involved at all stages of the design process. The sorts of artefacts that are generated during such processes reflect this collaborative way of working and will include things like white-board design sessions with post-it notes used to represent interface elements, textual narrative descriptions of things like domain information and scenarios, task analysis models, user descriptions and paper-based prototypes.

One of the problems we face when trying to capture UCD processes within a formal software engineering context is that the artefacts produced are intentionally informal. They aim to encourage users to feel able to participate and change the design, and lo-fidelity artefacts, such as paper prototypes for example, have been shown to be very successful for this purpose.

There have been several methods and tools developed which support prototyping or enable the use of tablet PCs [14], collaborative whiteboards [21] or desktop computers to generate prototypes in a manner similar to paper prototyping [8]. It may be that some, or all, of these tools could be adapted or extended to support the sort of work we are currently doing. However, as our focus is currently on existing commonly used design techniques and artefacts, we have deliberately chosen not to consider such tools here. Instead we focus on lo-fidelity artefacts like paper-based prototypes.

3 Formal Methods and Refinement

When we state that we wish to use formal methods as the basis for our system derivation we mean that we want to build models, at whatever level of abstractness/concreteness is most natural and useful to the developers of the system, which we can investigate with "mathematical" precision. So, typically, we want to build our models (write our specifications) in a language which has well-defined properties: syntax, semantics and logic. Without the first two properties we cannot (without a well-defined syntax) separate the specifications from all the other artefacts, or (without a well-defined semantics) know what a specification means even if we know, syntactically, that we have one.

The third requirement, that we have a logic, is also clearly necessary: being able to build a well-defined specification is a good start, but we also need to be able to precisely investigate that specification, see what its assumptions are, see what properties it has, see what implications for the system arise and so on. For all these necessary things we must have a logic.

So, our requirements are broad, not very onerous and leave developers open to choose whichever language they like to use (making decisions on grounds of familiarity, suitable for the task *etc.*) as long as it has our three properties.

3.1 Refinement

The idea behind refinement is very simple and goes back to Wirth [26]. It is based on the desire to be able to move between different models of a system without having any negative impact on a user's view or feel of the system in terms of its functionality or usability. As a simple example, we might move from a system that uses sets to one that uses arrays: here we move from abstract to concrete, from a convenient and useful idea (sets) to an implementation-oriented one (which probably includes too much detail).

This original idea behind refinement has been generalised so that we can think about not just differing implementations but differing levels of abstraction of model, from specification to implementation.

The basic intuition behind refinement is [9]:

> *Principle of Substitutivity*: it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken place. If a program can be acceptably substituted by another, then the second program is said to be a *refinement* of the first.

4 Integration of Techniques

Integration of different languages and models within formal methods is not unusual (indeed this activity has at least one whole conference devoted to it, namely IFM [13]). The central idea is to use the differing features and strengths of the different methods as appropriate. Sometimes it is enough to just use different formalisms to specify different parts or different properties of the system, but the best effect is seen when methods are fully integrated so there are formal links between them allowing for a fully rigorous development.

Our aim is to formally link the formal and informal processes so that we get all of the benefits of rigorous specifications and refinement, namely the ability to prove properties of a system and ensure formally that we meet requirements and end up with a correct implementation, while at the same time benefiting from the informal design methods of a UCD process which ensures we satisfy the user requirements and develop a usable interface.

Using formal methods in GUI design is not a new idea, and many different approaches to this have been taken. These may be along the lines of formalising particular parts of the design process, such as task analysis [19], or describing GUIs in a formal manner [10], or deriving implementations from formal models [11],[7]. However, what we are trying to do is to look at an existing design methodology, *i.e.* user-centred design, examine the types of processes and artefacts that are used and find ways of incorporating these into a formal process.

5 Presentation Model

The presentation model is used to formally capture the meaning of an informal design artefact such as a scenario, storyboard or prototype. It is a deliberately simple model because the informal artefacts it describes are themselves simple and easy to understand. This is important as it makes it easier to encourage others to adopt and use the model. When we talk about the *meaning* of a design artefact we are talking about what the UI described by the informal artefact is supposed to do, *i.e.* if it were transformed into an implementation what its behaviour would be. If we consider a paper-based prototype in isolation its meaning may be ambiguous; it requires some supporting information or context to make clear what is intended.

When a designer shows a prototype to a user, there is a discussion about what the prototype will do when the parts shown are interacted with. This forms what we call the narrative of the prototype, the accompanying story which allows the user to understand how it will work and what the various parts do. This allows a simulated interaction to take place which enables the user and designer to evaluate the suitability of the proposed design. The presentation model is a formal model which describes an informal design artefact in terms of the widgets of the design and captures their meaning. It is deliberately abstract and high-level. The presentation model is not intended to replace the informal design artefact, rather it acts as a bridge between the meaning captured by the design and the formal design process being used for the system functionality. The syntax for presentation models is given next.

5.1 Syntax

$$
\begin{array}{l}
\langle \mathit{pmodel} \rangle ::= \langle \mathit{declaration} \rangle \langle \mathit{definition} \rangle \\
\langle \mathit{declaration} \rangle ::= \mathit{PModel}\{\langle \mathit{ident} \rangle\}^+, \\
\qquad\qquad\qquad \mathit{WidgetName}\{\langle \mathit{ident} \rangle\}^+, \\
\qquad\qquad\qquad \mathit{Category}\{\langle \mathit{ident} \rangle\}^+, {}^{4} \\
\qquad\qquad\qquad \mathit{Behaviour}\{\langle \mathit{ident} \rangle\}^*, \\
\langle \mathit{definition} \rangle ::= \{\langle \mathit{pname} \rangle \mathit{is} \langle \mathit{pexpr} \rangle\}^+ \\
\langle \mathit{pexpr} \rangle ::= \{\langle \mathit{widgetdescr} \rangle\}^+ \mid \langle \mathit{pname} \rangle : \langle \mathit{pexpr} \rangle \mid \langle \mathit{pname} \rangle \\
\langle \mathit{pname} \rangle ::= \langle \mathit{ident} \rangle \\
\langle \mathit{widgetdescr} \rangle ::= (\langle \mathit{widgetname} \rangle, \ \langle \mathit{category} \rangle \ , (\{\langle \mathit{behaviour} \rangle\}^*)) \\
\langle \mathit{widgetname} \rangle ::= \langle \mathit{ident} \rangle \\
\langle \mathit{category} \rangle ::= \langle \mathit{ident} \rangle \\
\langle \mathit{behaviour} \rangle ::= \langle \mathit{ident} \rangle
\end{array}
$$

$\{Q\}^+$ *indicates one or more Qs*

$\{R\}^*$ *indicates zero or more Rs*

An example of a legal presentation model is then:

PModel	$p\ q\ r$
Widgetname	*aCtrl bCtrl cSel*
Category	*ActionControl SValSelector*
Behaviour	*dAction eAction fAction*

$$
\begin{array}{ll}
p\ \mathit{is} & (\mathit{aCtrl}, \mathit{ActionControl}, (\mathit{eAction}\ \mathit{fAction})) \\
 & (\mathit{bCtrl}, \mathit{ActionControl}, (\mathit{dAction})) \\
q\ \mathit{is} & (\mathit{cSel}, \mathit{SValSelector}, (\mathit{eAction}\ \mathit{fAction})) \\
r\ \mathit{is}\ p : q &
\end{array}
$$

This model describes a UI with two components, p and q (where these may be different windows, or different states of the UI). The entire UI (*i.e.* the combination of p and q) is described by r and the : operator acts as a composition. p has two widgets, *aCtrol* and *bCtrl*, which are both *ActionControls*. The behaviours associated with *aCtrl* are *eAction* and *fAction* and for widget *bCtrl* the associated behaviour is *dAction*. q has one widget, *cSel*, which is a *SValSelector* with the behaviours *eAction* and *fAction*. Presentation model r, therefore, is the combination of all of the widgets of p and q and describes the total possible behaviours of the UI.

[4] The categories used are based on the work in [2]

5.2 Semantics

We can now give the semantics of the model. Firstly, we can describe the complete model of a design as an environment ENV.

The environment is a mapping from the name (from the set *Ide* of identifiers) of some presentation model and its parts to their respective values:

$$\begin{aligned}&ENV = Ide \rightarrow Value\\&Value = Const + \mathbb{P}(Const \times Const \times \mathbb{P}\, Const)\\&Const = \{\overline{v} \mid v \text{ is an identifier}\}\end{aligned}$$

We use semantic functions to build up the contents of the environment and to describe its structure based on the given syntax.

$$\begin{aligned}&[\![\_]\!] : \langle pmodel\rangle \rightarrow ENV\\&Dc : \langle declaration\rangle \rightarrow ENV\\&Df : \langle definition\rangle \rightarrow ENV \rightarrow ENV\\&Expr : \langle pexpr\rangle \rightarrow ENV \rightarrow ENV\end{aligned}$$

$$[\![Decl\ Def]\!] = Df[\![Def]\!](Dc[\![Decl]\!])$$

$$\begin{aligned}&Dc[\![PModel\, \pi_1 \mathinner{..} \pi_n(_1)\ WidgetName\, \alpha_1 \mathinner{..} \alpha_n(_2)\ Category\, \epsilon_1 \mathinner{..} \epsilon_n(_3)\ Behaviour\\&\beta_1 \mathinner{..} \beta_n(_4)]\!] = \{\pi_i \mapsto \overline{\pi_i}\}_1^n(^1) \cup \{\alpha_i \mapsto \overline{\alpha_i}\}_1^n(^2) \cup \{\epsilon_i \mapsto \overline{\epsilon_i}\}_1^n(^3) \cup \{\beta_i \mapsto \overline{\beta_i}\}_1^n(^4)\end{aligned}$$

where $\{e_i\}_1^k$ is shorthand for the set $\{e_1, e_2, .., e_k\}$

$$\begin{aligned}&Df[\![D\ Ds]\!]\rho = Df[\![Ds]\!](Df[\![D]\!]\rho)\\&Df[\![P\ is\ \psi]\!]\rho = \rho \oplus \{P \mapsto Expr[\![\psi]\!]\rho\}\end{aligned}$$

where ρ represents the current environment.

$$\begin{aligned}&Expr[\![E\ Es]\!]\rho = Expr[\![E]\!]\rho \cup Expr[\![Es]\!]\rho\\&Expr[\![\psi : \phi]\!]\rho = Expr[\![\psi]\!]\rho\ \cup\ Expr[\![\phi]\!]\rho\\&Expr[\![(N\ C\ (b_1 \mathinner{..} b_n))]\!]\rho = \{(\rho(N)\,\rho(C)\,\{\rho(b_1) \mathinner{..} \rho(b_n)\})\}\\&Expr[\![I]\!]\rho = \rho(I)\end{aligned}$$

Our presentation models consist of widgets with names, categories and behaviours. In our semantics we have shown how the syntax of the model creates mappings from identifiers to constants in the environment (which represents the design that the model is derived from). The presentation model semantics is a conservative extension of set theory, that is, everything which is provable about presentation models from the semantics is already provable in set theory using the definitions given in the semantic equations. This then allows us to rely on the existing sound logic of set theory to derive a necessarily sound logic for our presentation models.

Next we provide an example of a UI design and presentation model of that design which we will use to illustrate the uses and extensions for presentation models.

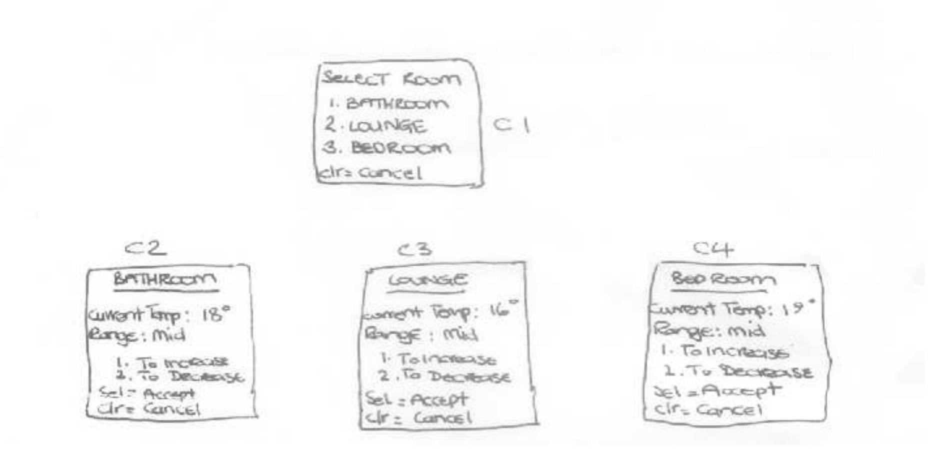

Fig. 1. Design for Mobile Phone Application UI

6 Example

The following example is an adaptation of an example given by Calvery *et al.* in [5] and [4]. The example involves a home heating control system which is accessible via several different devices, namely a home-based, wall-mounted control panel, a web-server running on a standard PC, a PDA and a WAP-enabled mobile phone. The control system supports the monitoring and control of temperatures in a number of different rooms as well as overall adherence to ambient temperature levels. For the purposes of our example we use an amended version of the mobile phone application UI which allows us to illustrate our particular points.

A proposed UI design for the mobile phone version of the system is given in Figure 1. This shows the four different screens which make up the UI for the application which we label $C1$, $C2$, $C3$ and $C4$ respectively. The presentation model for the mobile phone UI design follows (some detail has been omitted for brevity):

PModel	*MPHeat MPMenu MPBed MPLounge MPBath*
Widgetname	*BathSelect LoungeSelect BedSelect QuitOpt IncBathOpt DecBathOpt IncLoungeOpt DecLoungeOpt IncBedOpt DecBedOpt AcceptOpt CancelOpt BathTempDisp BathRangeDisp LoungeTempDisp LoungeRangeDisp BedTempDisp BedRangeDisp*
Category	*ActCtrl SValSel SValRespdr*
Behaviour	*ShowBath ShowLounge ShowBed QuitApp IncBathTemp DecBathTemp IncLoungeTemp DecLoungeTemp IncBedTemp DecBedTemp StoreSettings ShowMenuPage DispBathTemp DispBathRange DispBedTemp DispBedRange DispLoungeTemp DispLoungeRange*

MPMenu is *(BathSelect,ActCtrl,(ShowBath))*
(LoungeSelect, ActCtrl,(ShowLounge))

	(BedSelect, ActCtrl,(ShowBed))
	(QuitOpt, ActCtrl,(QuitApp))
MPBed is	*(BedTempDisp, SValRespndr,(DispBedTemp))*
	(BedRangeDisp, SValRespndr,(DispBedRange))
	(IncBedOpt, SValSel,(IncBedTemp))
	(DecBedOpt, SValSel,(DecBedTemp))
	(AcceptOpt, ActCtrl,(StoreSettings))
	(CancelOpt, ActCtrl,(ShowMenuPage))
MPLounge is	... omitted
MPBath is	... omitted
MPHeat is	*MPMenu : MPBath : MPLounge : MPBed*

7 Using the Presentation Model

7.1 Presentation Models and Refinement

Our first use for the presentation model is to enable us to include the design of the UI in our formal refinement process. We have previously given a detailed account of this process [3] and it is not our intention to repeat these details here. However we will give an outline of the process and direct the interested reader to [3].

We have talked about a relationship between the activities of our formal and informal design processes. We start to define this at the first design activity for each method, *i.e.* requirements gathering for the formal specification and determining user requirements for the UI design. Rather than treating these two activities independently we need to ensure that the information gathered from each is used together to produce a specification and UI requirements which are not only fully inclusive, but which share a vocabulary and compliment each other. We create a formal specification for a system and use naming conventions which indicate which of the given operations are user operations, that is, which of the operations upon the system state should be made available directly to the user via the UI.

If we were to create a formal specification for the heating application, based on the requirements given in [5] and using the specification language Z [1], we would follow this convention and name the operations we describe accordingly. For example, in order to describe the requirements of a user to be able to increase the temperature in any of the rooms, we would expect to see the following in our specification (using the Z idiom of *promotion*):

$$
\begin{array}{|l}
\textit{Room} \\
\hline
\textit{CurrentTemp} : \textit{TEMP} \\
\hline
\textit{CurrentTemp} \leq \textit{MaxTemp} \\
\textit{CurrentTemp} \geq \textit{MinTemp} \\
\hline
\end{array}
$$

$$\textit{TempControlSystem} \mathrel{\widehat{=}} [\textit{rooms} : \textit{RID} \rightarrowtail\!\!\!\!\!\!\rightarrow \textit{Room}]$$

$$
\begin{array}{|l}
\hline
\Phi\, UpdateRoomTemp \\
\Delta\, TempControlSystem \\
\Delta\, Room \\
rid? : RID \\
\hline
rid? \in \operatorname{dom} rooms \\
\Theta Room = rooms\ rid? \\
rooms' = rooms \oplus \{rid? \mapsto \Theta Room'\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
IncreaseRoomTemp \\
\Delta\, Room \\
newTemp? : TEMP \\
\hline
newTemp \geq MinTemp \\
newTemp \leq MaxTemp \\
CurrentTemp' = newTemp \\
\hline
\end{array}
$$

$$
\begin{array}{l}
USER\_IncRoomTemp \mathrel{\widehat{=}} \exists\, \Delta Room \bullet \Phi\, UpdateRoomTemp \wedge \\
\qquad\qquad\qquad IncreaseRoomTemp
\end{array}
$$

As part of our refinement process we need to ensure that all user operations described in the specification have been described in the UI design. That is, we should ensure that the system informally described by the GUI design is a refinement of the specification. From the presentation model of the design we can produce a Z description (using the framework for describing widget categories in Z given in [2] as the basis). From the presentation model of the mobile phone heating application we can derive such a Z description. If we focus on the requirement to increase the temperature of the bedroom we can look at one part of the derived Z description which is:

$$
\begin{array}{|l}
\hline
IncBedTempOp \\
\Delta\, Bedroom \\
iActValue? : TEMP \\
iAction? : ACTION \\
\hline
iAction? = IncBedTemp \Rightarrow \\
\qquad bedTemp' = iActValue? \\
iAction? \neq IncBedTemp \Rightarrow \\
\qquad bedTemp' = bedTemp \\
\hline
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{IncBedTempSelCtrl}} \\
\mathit{iCState} : \mathit{CONTROLSTATE} \\
\mathit{iSelValue} : \mathit{TEMP} \\
\mathit{iAction}! : \mathit{ACTION} \\
\mathit{iActValue}! : \mathit{TEMP} \\
\hline
\mathit{iCState} = \mathit{Active} \Rightarrow \\
\qquad \mathit{iAction}! = \mathit{IncBedTemp} \\
\mathit{iCState} = \mathit{NotActive} \Rightarrow \\
\qquad \mathit{iAction}! = \mathit{NoAction} \\
\mathit{iActValue}! = \mathit{iSelValue} \\
\hline
\end{array}$$

$$\begin{array}{l}
\mathit{ActiveIncBedTemp} \mathrel{\widehat=} [\mathit{IncBedTempSelCtrl} \mid \mathit{iCState} = \mathit{Active}] \\
\qquad\qquad \gg \mathit{IncBedTempOp}
\end{array}$$

The presentation model for this example describes a widget *BedTempDisplay* whose category is *SValResponder*. If we refer to [2] we see that to describe such a widget in Z we must provide a schema with observations on active state (which captures the notion of user interaction), selected values and behaviours, and then link this (using the Z piping notation) to an operation schema which describes the associated behaviour (in this case setting the temperature to a new value).

We can now use the standard simulation techniques (which are based on [27] and [9]) to show that a refinement holds between the UI design and the specification.

7.2 Presentation Models and Design Equivalence

Following on from the use of presentation models in refinement we have derived a notion of equivalence between designs, based again on the presentation model. The intention here is to be able to take different UI designs (for the same system) and using the presentation models of these designs determine if they can be considered in some way equivalent.

Design equivalence is important during a refinement process, where rapid iteration of designs means it may be more practical to require proof of refinement back to the specification only when the design changes significantly, *i.e.* when it is no longer functionally equivalent to the previous version of the design. We define this notion of functional equivalence next.

The functionality of a design is given by the set of behaviours of the presentation model of that design. So if we wish to compare two different UI designs to determine whether or not they have the same functionality, then we can simply compare the corresponding behaviour sets of their presentation models. Formally we state:

Definition 7.1 If *DOne* and *DTwo* are UI designs and *PMOne* and *PMTwo* are their corresponding presentation models then:

$$\begin{array}{l} DOne \equiv_{func} DTwo =_{df} Behaviours[PMOne] = Behaviours[PMTwo] \\ Behaviours[P] =_{df} \{[\![P]\!]b \mid b \in act(P)\} \end{array}$$

$act(P)$ is a syntactic function that returns all identifiers for behaviours in P.

Design equivalence is also important in cases where we are designing several interfaces for different versions of a system, as we are in the home heating system example. We want to be sure that the different versions of the system provide the user with the same functionality. Again, we could prove this by using refinement techniques from each of the different system designs back to the specification. However, design equivalence provides a weaker approach to refinement which allows us to ensure that the intended behaviour (both system functional behaviour and UI functional behaviour) is provided by all of the UIs.

As well as functional equivalence we have considered other types of equivalence which exist between designs, namely *Component Equivalence* and *Isomorphism.* We will not go into the details of these types of equivalence here as they are beyond the scope of this paper.

7.3 Presentation Models and Design Consistency

The third use of presentation models we present here is their use in ensuring consistency between designs. Consistency is an important principle of UI design. Shneiderman [22] includes consistency as one of his eight golden rules for interface design:

Strive for consistency.
Consistent sequences of actions should be required in similar situations;
identical terminology should be used in prompts, menus, and help screens;
and consistent commands should be employed throughout.

An application may consist of numerous different screens and dialogues, so maintaining consistency throughout is not a trivial task. One of the things we can ensure, using the presentation model, is that controls which have the same function have the same name (so the user does not have to remember that in one part of the interface they use *Quit* to exit the interface and in another they use *Close*). Conversely we can also check, again using the model, that controls with the same name have the same function and this ensures that the user always knows what to expect when they encounter such a control.

8 Limitations and Extensions

We have provided some examples of how we can use presentation models of informal designs to not only help with our aim of integration of informal design artefacts into our formal process (via the refinement mechanism and equivalence), but also in dealing with design concerns such as consistency.

There is, however, more to say on the subject of refinement. While we may be able to prove (or disprove) that particular functionalities are included in a UI design (because they are a behaviour of one of the widgets) this is not enough to imply refinement.

As we have already stated, a UI may consist of many different windows and dialogues. The mechanics of moving between these different windows or dialogues are included in the UI-functionality of the design (these are the behaviours which do not correspond to underlying system functions, but instead are used to change the state of the UI). For a refinement to hold between a specification and a design we need to ensure that not only do the required system functions exist in some part of the UI design, but also that they are reachable via some UI function.

Proving the property of reachability in a UI is a common concern in much of the early work on using formal methods with UIs. Rather than trying to adapt and incorporate an existing technique for this into our process, we want to be able to use our presentation model for this purpose also.

The problem with trying to capture the idea of dynamic change of the UI via the presentation model is that that the model gives us a static view of the design. It describes a total environment given by the design (which we can consider to be all of the possibilities of that design), but the (deliberately) simple use of a triple for each widget does not hold enough information to extend its use to dynamic behaviour. One possible solution to this would be to extend the model with additional information. However, we want to avoid making it so complex that it becomes a burden upon designers or formal practitioners to learn and use. We have decided to use another common formalism, in conjunction with the presentation model, in order to be able to prove these more dynamic properties. The formalism that we have chosen is that of Finite State Machines (FSM).

FSM have been used previously for GUI modelling in both design (as early as the late 1960's [16]) and as a way of evaluating interfaces [15]. One of the drawbacks with using FSM in this way is the known problem of state explosion, where the number of states of the machine becomes intractably large. Given the complexity of modern UIs this is certainly a concern and potential problem whenever we try and use FSM to model GUIs or GUI behaviour. However, because we already have an abstraction of the UI (the presentation model) we can use this in conjunction with a FSM and in most cases we produce a FSM which require only a very small number of states. We produce a FSM which is at a high level of abstraction and decorate it with presentation models which provide the lower-level meaning.

Our FSM consists of: a finite set of states, Q; a finite set of input labels, $\sum$; a transition function, δ, which takes a state and an input label and returns a state ($q \rightarrow a \rightarrow q'$); a start state, q_0, one of the states in Q. The FSM is then a four tuple $(Q, \sum, \delta, q_0)$.

Each of the states in Q is associated with the name of a presentation model in the overall model which the FSM describes. When the FSM is in a particular state then the presentation model associated with that state is the currently active one, *i.e.* the part of the UI described in that model is visible to the user and available

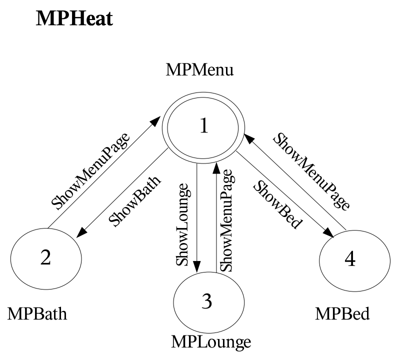

Fig. 2. PIM for MPHeat Presentation Model

for interaction. We can clearly extend the definition of FSM to a quintuple and add a mapping from state names to presentation model names to formally show this association. The input labels in $\sum$ are themselves the names of behaviours taken from the behaviour sets of the presentation models. In this way we can associate the UI functionality of parts of the design with the dynamic behaviour which makes available different parts of the interface to the user. We call the combination of presentation model and FSM in this way a presentation and interaction model (PIM).

We give a definition of well-formedness for our FSM as follows:

A PIM of a presentation model is well-formed iff the labels on transitions out of any state are the names of behaviours which exist in the behaviour set of the presentation model which is associated with that state.

Using the notation for our FSM we can give this more formally as:

$$\forall (q, t, q') : \delta \bullet \exists\, b \in act(q_{PModel}) \bullet t = b$$

where q_{PModel} is the presentation model associated with state q.

If we return again to the design of the mobile phone-based application for the heating example, given in Figure 1, we can see that before we can consider the reachability of functions of this UI we need to capture the way in which we move from one part of the interface to another. It is common for prototypes to be annotated to include this sort of information (or in the case of storyboards this is implicit in the flow of the diagrams).

We capture this implied dynamic behaviour between parts of the UI using a FSM which we decorate with parts of the presentation model to give meaning. Figure 2 gives the PIM for the design of Figure 1.

Now, in order to show that a particular behaviour is reachable we first need to show that the part of the UI it is in (*i.e.* the component presentation model which includes this behaviour in its set of behaviours) is itself reachable in the FSM, and this can be shown using standard FSM methods.

9 Conclusion

In this paper we have described how to aid the integration of formal methods with informal UI design methods. This approach involves creating a formal model of informal design artefacts in a way which allows us to then use them in formal processes.

We have described the presentation model, which formally captures an informal UI design, and discussed how we can use this to include designs in a formal refinement process as well as for design equivalence and consistency checking. The presentation model allows us to capture static properties of a UI design and we have subsequently shown how we can use this with another formalism, FSM, to capture dynamic UI behaviour based on UI functions which change the available functionality of the UI for a user, giving PIMs.

The main advantage we propose for the presentation model and the methods we have shown is that they work in conjunction with existing methods being used by formal practitioners and designers. We do not require that these groups abandon their existing methods and techniques, but rather enhance these with a relatively straight-forward formalism and set of techniques which work alongside, rather than replace, their existing methods.

This paper is designed to give an overview of our work and techniques, rather than going into detail about one particular part of it. We have described our general work in this area and where appropriate we have referred the reader to more detailed accounts in prior publications.

Considerably more work has been done in the area of formal methods and UI design than we are able to give account of in this paper. We have tried to reference appropriate works, and explain the difference in our approach where relevant, but these references should by no means be considered an exhaustive list.

References

[1] ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics.* Prentice-Hall International series in computer science. ISO/IEC, first edition, 2002.

[2] Judy Bowen. Formal specification of user interface design guidelines. Masters thesis, Computer Science Department, University of Waikato, 2005.

[3] Judy Bowen and Steve Reeves. Formal refinement of informal GUI design artefacts. In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*, pages 221–230. IEEE, 2006.

[4] G. Calvary, J. Coutaz, and D. Thevenin. Supporting context changes for plastic user interfaces: A process and a mechanism. In A. Blandford, J. Vanderdonckt, and P. Gray, editors, *Joint Proceedings of HCI'2001 and IHM'2001*, pages 349–363. Springer-Verlag, 2001.

[5] Gaelle Calvary, Joèlle Coutaz, and David Thevenin. A unifying reference framework for the development of plastic user interfaces. In *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, pages 173–192, London, UK, 2001. Springer-Verlag.

[6] Francesco Correani, Giulio Mori, and Fabio Paternò. Supporting flexible development of multi-device interfaces. In *EHCI/DS-VIS*, pages 346–362, 2004.

[7] Antony Courtney. Functionally modeled user interfaces. In *Interactive Systems. Design, Specification, and Verification. 10th International Workshop DSV-IS 2003, Funchal, Madeira Island (Portugal) J. Joaquim, N. Jardim Nunes, J. Falcao e Cunha (ed.)*, pages 107–123. Springer Verlag Lecture Notes in Computer Science LNCS, 2003.

[8] Adrien Coyette, Stéphane Faulkner, Manuel Kolp, Quentin Limbourg, and Jean Vanderdonckt. Sketchixml: towards a multi-agent design tool for sketching user interfaces based on usixml. In *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*, pages 75–82, New York, NY, USA, 2004. ACM Press.

[9] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Formal Approaches to Computing and Information Technology. Springer, May 2001.

[10] David J. Duke, Bob Fields, and Michael D. Harrison. A case study in the specification and analysis of design alternatives for a user interface. *Formal Asp. Comput.*, 11(2):107–131, 1999.

[11] D. F. Gieskens and J. D. Foley. Controlling user interface objects through pre and postconditions. In *Proc. of CHI-92*, pages 189–194, Monterey, CA, 1992.

[12] A. Hussey, I. MacColl, and D. Carrington. Assessing usability from formal user-interface designs. Technical Report TR00-15, Software Verification Research Centre, The University of Queensland, 2000.

[13] IFM05. http://www.win.tue.nl/ifm/, 2005.

[14] J. Landay. Silk: Sketching interfaces like krazy. In *Human Factors in Computing Systems (Conference Companion), ACM CHI '96, Vancouver, Canada, April 13–18*, pages 398 – 399, 1996.

[15] A. Paiva, N. Tillmann, J. Faria, and R. Vidal. Modeling and testing hierarchical GUIs. In *D. Beauquier, E. Borger, and A. Slissenko, editors, ASM05*, pages 329–344. Universite de Paris, 2005.

[16] David L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 1969 24th national conference*, pages 379–385. ACM Press, 1969.

[17] F. M. Paternò, M.S. Sciacchitano, and J. Lowgren. A user interface evaluation mapping physical user actions to task-driven formal specification. In *Design, Specification and Verification of Interactive Systems*, pages 155—173. Springer Verlag, 1995.

[18] Fabio Paternò. Task models in interactive software systems. *Handbook of Software Engineering and Knowledge Engineering*, 2001.

[19] Fabio Paternò. Towards a UML for interactive systems. In *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, pages 7–18, London, UK, 2001. Springer-Verlag.

[20] G. E. Pfaff. *User Interface Management Systems.* Springer-Verlag New York, Inc., 1985.

[21] Beryl Plimmer and Mark Apperley. Computer-aided sketching to capture preliminary design. In *CRPIT '02: Third Australasian conference on User interfaces*, pages 9–12, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[22] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison Wesley Longman Inc, 3rd edition, 1998.

[23] Graeme Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.

[24] H. Thimbleby. Design of interactive systems. *The Software Engineer's Reference Book*, 1990.

[25] Harold Thimbleby. User interface design with matrix algebra. *ACM Trans. Comput.-Hum. Interact.*, 11(2):181–236, 2004.

[26] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[27] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.