Innovations Syst Softw Eng (2008) 4:125–141
DOI 10.1007/s11334-008-0049-0

ORIGINAL PAPER

Formal models for user interface design artefacts

Judy Bowen · Steve Reeves

Received: 1 October 2007 / Accepted: 28 February 2008 / Published online: 16 April 2008

Abstract There are many different ways of building software applications and of tackling the problems of understanding the system to be built, designing that system and finally implementing the design. One approach is to use formal methods, which we can generalise as meaning we follow a process which uses some formal language to specify the behaviour of the intended system, techniques such as theorem proving or model-checking to ensure the specification is valid (i.e., meets the requirements and has been shown, perhaps by proof or other means of inspection, to have the properties the client requires of it) and a refinement process to transform the specification into an implementation. Conversely, the approach we take may be less structured and rely on informal techniques. The design stage may involve jotting down ideas on paper, brainstorming with users etc. We may use prototyping to transform these ideas into working software and get users to test the implementation to find problems. Formal methods have been shown to be beneficial in describing the functionality of systems, what we may call application logic, and underlying system behaviour. Informal techniques, however, have also been shown to be useful in the design of the user interface to systems. Given that both styles of development are beneficial to different parts of the system we would like to be able to use both approaches in one integrated software development process. Their differences, however, make this a challenging objective. In this paper we describe models and techniques which allow us to incorporate informal design artefacts into a formal software development process.

J. Bowen (✉) · S. Reeves
Department of Computer Science,
University of Waikato, Hamilton, New Zealand
e-mail: jab34@cs.waikato.ac.nz

Keywords Formal methods · User-centred design · Refinement · User interfaces

1 Introduction

Software development is an increasingly complex task, with many different, and often conflicting, requirements which must be satisfied (such as the requirement to produce the software as quickly as possible and as cheaply as possible but with high end-user expectations). We want the software that we build to be both correct and robust, that is we want to be sure that it not only does the right thing, but that it does so under all circumstances. Sometimes, with safety-critical software for example, it is not enough to feel certain that our system designs are correct, we require proof of this before we implement them. These are just some of the reasons we rely on formal methods when we develop software. When using such methods we follow procedures which ensure we correctly understand what the requirements of the software are, that we design it in such a way that these requirements will always be met, and we transform our designs into implementations in a way which preserves these guarantees. While we may believe that developing software in this way is enough to ensure we meet our intended aims, there is another equally important concern, that of the user. If a user is unable to satisfactorily use the software we have built, then despite the correctness of the underlying application we have failed to meet our objectives.

A different approach to software development, which takes these concerns regarding the user on board, is that of user-centred design (UCD). UCD techniques make the users central to the development process and keep them involved at all stages of design and development to ensure that the systems we build meet their expectations and are usable by the people who will use the system. While we refer to UCD techniques as informal, this is not to suggest that they are unstructured or ad hoc, but rather that the artefacts they produce are themselves informal. Such artefacts may include the results of ethnographic studies, brain-storming sessions using white-boards and post-it notes, paper and pencil sketches, etc. and are intended to convey information quickly and easily to non-technical people, i.e., real users rather than software developers. They are intended to be discussed and used as a starting point for design decisions, and as such they may be ambiguous or unclear if examined in isolation from the rest of the design process.

As a reflection of the different types of approaches, it is not unusual for software design to be tackled 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 user interface (UI) 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 is exemplified by the Seeheim model [30]. 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. However, there are also some problems associated with separating our software and developing parts of it using different methods. 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. Separating parts of the system out and designing them using informal techniques means that when we integrate everything into one final implementation, any guarantees we may have had regarding correctness (from our formal process) will be lost as we will have no such guarantees about the other parts of the system. In fact we cannot even be sure that the different parts of the system have been designed with the same end goal in mind as there is no common blueprint to both parts of the design perhaps due to the difference between the "languages" of the designers and the techniques they use.

With this in mind a need to somehow bring these two types of process closer together has been identified. The gap between the formal and informal has been discussed many times, notably in 1990 by Thimbleby [36]. Many different approaches have been taken over recent years by different groups of researchers to try and bridge this gap. Such work may fall into one of the following categories:

– Development of new formal methods for UI design. e.g., Modelling UIs using new formalisms [9];
– Development of hybrid methods from existing formal methods and/or informal design methods. e.g., using temporal logic in conjunction with interactors [29];
– Use of existing formal methods to describe UIs and UI behaviour. e.g., matrix algebra for UI design [37];
– Replacing existing human-centred techniques with formal model-based methods. e.g., using UI descriptions in Object-Z [34] to assess usability [21].

Such work is demonstrably a step forward in bringing together formal methods and UI design. However, the methods and techniques which have been developed have failed, in the most part, to become mainstream. Those which have become mainstream (such as task analysis methods) are generally focussed on one small part of the design stage rather than the complete process.

One of the reasons for this seeming reluctance for either group (those involved in user interface design and formal methods practitioners) 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. While we take the stance that formal methods are important, and necessary, when building software, we appreciate that UI designers may be graphic designers or usability experts who are comfortable with the methods they currently use rather than with using notations based on mathematics and logic.

The approach we are taking then is that rather than trying to change the methods used by different groups of software developers, we will instead consider the existing, diverse, methods being used and develop ways of formally linking them together. We do not assume that any one particular formal approach is being used (i.e., we do not restrict ourselves to trying to work with just one particular formal notation or process), just that there is some formal approach. Similarly we recognise the diversity of methods and techniques used to design UIs and take a general approach to the artefacts produced. Our only requirement is that, again, a UCD approach is being taken rather than just the random generation of the UI (by, for example, just hacking code). So, we intend to find ways of interpreting the sorts of informal design artefacts produced in a UCD process within a formal framework.

In [5], we provided an introduction to two models, presentation models and presentation interaction models (PIMs) which are designed to formally describe informal design artefacts. In this paper we give an expanded description of these models, including a more detailed description of the context of the research behind them. We will further show that in addition to the benefits we outlined in [5] in ensuring consistency and correctness between the formal and informal processes, we can also use these models to guide the design of the UI and consider properties of the design relating to usability.

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. They aim to find out about the tasks the users need to perform with the software, as well as understanding the background to those requirements, such as current working practices of the users, their experience with similar software, internal company working processes that will be affected by this new software, etc. In this way the designer is able to build up a picture of how the software will fit into the work environment, who the people are who will use the software and how they envisage using it.

When we talk about following a UCD process we do not mean that there is a set of defined methods which must be followed in a particular order, but rather that we adopt any number of techniques (from those considered to be UCD techniques) which allow us to gather the sort of information we have described above. These may include things like ethnographic studies, which are used very early on in the process to find out about the users and their work environment. Ethnographic studies are recognised as a useful way to gather comprehensive background information, but it is also recognised that the volume and nature of the data gathered can make it hard to know how to apply it within the design process [14].

Subsequently we may use 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., [8,27]. The benefits of such models are that they allow us to use task information in a structured manner within the design process and also consider the effects of our design choices in terms of sub-tasks and complexity. However, such methods focus on just one small part of the overall UI design rather than the whole process.

UCD practitioners may also use scenarios and personas to enhance the task analysis process and give details of specialised requirements and user behaviours. These expand on task analysis by examining "what if" cases, e.g., "What if a person like X wants to do Y?". Again the information gathered in this way is often in the form of a narrative or a summary of points and so must be interpreted by designers in order to make use of it.

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. While we may describe such artefacts as informal and simple, they are intentionally so. They are easy points of reference for communicating with users (in that they do not rely on specialist knowledge or design skills) and their informality encourages users to feel a part of the process and able to make changes or suggestions. Lo-fidelity artefacts, such as paper prototypes for example, have been shown to be very successful for this purpose.

Several methods and tools have been developed which support prototyping or to enable the use of tablet PCs [23], collaborative whiteboards [31] or desktop computers to generate prototypes in a manner similar to paper prototyping [10] but which generate a computer-based design. 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 on existing commonly used design techniques and artefacts, we have deliberately chosen not to consider such tools here. Just as the way in which a UCD practitioner's approach to their task may be different from someone else engaged in a similar process, so the prototypes or design artefacts, and the way we produce them, may vary. The point is, however, that the underlying structure of such artefacts is not formally defined whichever set of techniques is used.

So, given that we have diverse and informal artefacts we are faced with the challenge of how to use them within a formal process. In the next section we describe what we mean by a formal software development process and then move on to show how our research allows us to begin integrating the formal and informal.

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.

The idea behind refinement is very simple: it is a structured progression from some abstract version of a system towards a more concrete version. It is based on the desire to be able to move between different descriptions and/or implementations of a system (perhaps because underlying software changes, or hardware changes, or better versions become available or are developed, and so on) without having any negative impact on a user's view or feel of the system in terms of its functionality or usability (though they might, as might be intended, see that it runs faster, for example). This original idea has been generalised so that we can think about not just differing implementations but differing levels of abstraction from specification to implementation.

The basic intuition behind refinement is [11]:

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.

So, if a user is used to using a certain sequence of actions $\langle a_0, a_1, \ldots, a_n \rangle$ (i.e., using the facilities provided by a certain program P_0 which implements the actions a_i) then if we change the program to P_1 and the user carries out their sequence of actions again and gets the same behaviour from P_1 as they got from P_0, then we would say that P_1 is a refinement of P_0.

Note that this story allows P_1 to provide more facilities than P_0—but the point is that the user, when carrying out their sequence of actions, cannot tell. The change in underlying program has in no way degraded the system as far as our user is concerned. It may have improved the system, by supplying more facilities, but our user will not be able to tell (though of course we are likely to let them know of the new facilities—and probably extract some money for them!).

Another way a refinement can acceptably change a program is by reducing any nondeterministic behaviour that it might have. So, if due to invoking action a program Q_0 sometimes exhibits behaviour b_0 and sometimes behaviour b_1 and the user does not care which of these behaviours a causes, then replacing program Q_0 with program Q_1 which always exhibits behaviour b_0 when action a is invoked will not matter to the user (and will not be detectable by them—to be sure we had removed the nondeterminism the user would have to run an infinite number of experiments to check that b_1 was never exhibited, which of course they could never complete and so they could never be completely sure—having a suspicion is not enough!). So, Q_1 would be a refinement of Q_0.

This sort of refinement is a useful idea when we are building prototypes and then using them as a basis for moving to a final implementation. If some aspects of the prototype "do not matter", e.g., what happens when action a is invoked is of no interest to the user, then the prototype can do anything (so probably something simple and easy to code) due to action a. Later, we can refine away (probably in several steps) this lack of interest so that action a, which now might "matter", does what we require.

Another informal view of refinement (which turns out to be formally equivalent to the view given above) is to think of a system as a kind of contract between provider and consumer [24], with the emphasis on what works for the consumer, so, for example, if contract P_1 is no worse than contract P_0 for the consumer then P_1 refines P_0. Morgan gives examples like: a 220/110 V outlet refines a 220 V outlet; a watch water-resistant to 50 m refines a splash-proof one; a program that needs at least 2 Mb refines one that needs at least 4 Mb.

These ideas grew out of a desire to completely, precisely and unambiguously describe one of the first general methods for dealing with the construction of complex software, i.e., stepwise refinement (Wirth [40] is a classic reference here, but see our further discussion below too).

The reason for studying refinement is to understand it as fully as possible (i.e., all its implications, pitfalls, features). The reason for formalising it is, as ever with formalisation, to allow it to be precisely and unambiguously described, and hence to allow it to form the basis of a precise, unambiguous method (i.e., a formal method) for moving from a description at some level of abstraction towards an implementation (i.e., a less abstract description). One other reason for formalising is, of course, to allow ourselves to develop tools (i.e., pieces of software) which will support us in the process of refinement, and since any piece of software is itself a formal artefact, having a formalisation of a process gets us a good way towards having some software to support the process.

So far, we have talked about moving from one program to another in one "move" or, as we shall say, *step* of refinement.

A requirement on such steps is that each should be, in some sense, understandable: ideally, we would look at a step and have a story to tell about it in the vocabulary of the application area or, if we are close to implementation, in terms a programmer could understand and appreciate. This will tend to mean, so as to be intellectually manageable, that each step is modest in size and ambition. In any real, large piece of software development, then, the journey from initial description (probably a specification) to implementation is likely to be made of many refinement steps—again our ideal would be that the sequence of steps tells an understandable story about moving from specification to implementation. This sequence is what gives rise to the term *stepwise refinement* when we talk about this mode of software development.

Of course, this term can refer (and historically this was the primary notion with the work of Dijkstra [12]) to a more-or-less informal process, as well as to the more formal notion we have presented above. However, wherever on the spectrum of formality our process appears, the central requirement is that development proceeds by a sequence of steps and each step should move us towards implementation in a way that is clear and understandable.

Once a notion of refinement is in place we would want to check that it has certain properties. For example, the most important property is transitivity. This allows us to move stepwise towards an implementation with each subsequent refinement preserving earlier refinements. Another important property that would probably be required is that of *monotonicity*. That is, if we have a complex system specified by joining together (in some way) several partial systems (as we usually do via a *divide-and-conquer* strategy for dealing with large systems) we want to be sure that if one of the parts of the system is refined then the version of the system formed using the same constructions as the original system for joining together the original parts together with the new refinement of one of the parts is also a refinement of the original system. That is, by refining a part of the system we want to be sure we have then refined the whole.

This guarantees, amongst other things, that the refinement of part of the system has not interfered with or otherwise compromised the rest of the system, which has not been changed. If this property holds then we say the refinement relation is monotonic. Clearly, monotonicity is important if we want to gradually refine the system (by refining it in parts) into an implementation. Otherwise, we would have to deal with the whole system at each refinement step, increasing the likelihood of error and that we succumb to the complexity of having to reason about a large and complex system. This is, of course, the usual argument for modular construction of software for dealing with complexity.

There is much more to say about refinement (it has a long and influential history even if we restrict our attention to the more formal approaches) but we shall now move our attention to the application of refinement to our particular problem and leave the interested reader to consult the texts we have referred to above.

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 [22]). 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 used within modular software development 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 transform these into 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 UI 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 [28], or describing UIs in a formal manner [15], or deriving implementations from formal models [9,16].

One important difference between our work and previous research into formal methods and UI design is that we are not trying to formalise the process of UI design itself, but rather we want to find ways of formally capturing the information produced by an informal UI design process. As we will show in the next sections, using the models we have derived we are able to do things like specify UI behaviour, but this is not driven by the formal process, it is another way of viewing the information generated informally and we use the two in conjunction with each other.

5 Presentation model

The first model we describe is called the presentation model. It is used to formally capture the meaning of an informal design artefact such as a scenario, storyboard or prototype. It is a deliberately very 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 we say 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 interactive components of the design (which we will refer to as widgets) and captures their meaning. That is, it formally describes the narrative of the design artefact. It is deliberately abstract and high-level and 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. By describing the intended behaviour of the design artefact it removes the ambiguity that may exist when the design is considered by itself. The formal structure of the presentation model gives us a different view of the design and enables us to consider it within our formal framework.

5.1 Syntax

We start by giving the syntax of presentation models.

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

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

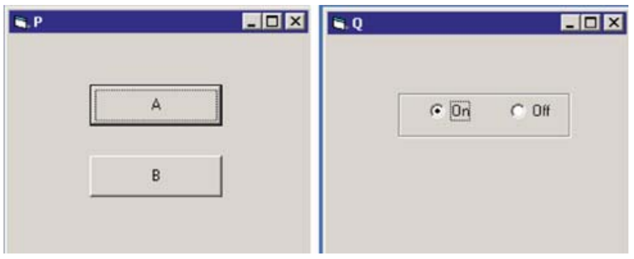

Fig. 1 Example UI design

Each presentation model begins with a set of declarations which introduces the identifiers for all parts of the model. Then we describe the design in terms of a set of widget descriptions which are given in a tuple consisting of an identifier, a category and a set of behaviours. The widget categories used are taken from the work described in [4] which provides hierarchies of widgets based on their high-level behaviour (for example, do they cause actions to occur when a user interacts with them, or do they provide information back to the user?) Where a widget has more than one behaviour associated with it the meaning intended is that the widget does more than one thing, it does not indicate choice. We assume that the designs we are modelling are deterministic on the basis that nondeterminism in a UI design either indicates that there are parts of the design not yet complete, and which can therefore be omitted from the model, or that there is confusion about what a particular widget will do, which we expect to resolve either prior to, or during, the building of the presentation model.

An example of a syntactically correct presentation model is then:

$$
\begin{array}{ll}
PModel & p\ q\ r\\
Widgetname & aCtrl\ \ bCtrl\ \ cSel\\
Category & ActionControl\ \ SValSelector\\
Behaviour & dAction\ \ eAction\ \ fAction
\end{array}
$$

$$
\begin{array}{l}
p\ is\quad (aCtrl, ActionControl, (eAction\ \ fAction))\\
\qquad\quad (bCtrl, ActionControl, (dAction))\\
q\ is\quad (cSel, SValSelector, (eAction\ \ fAction))\\
r\ is\ p : q
\end{array}
$$

This model describes a UI such as that given in Fig. 1. It has 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, $aCtrl$ 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 indicates the total possible behaviours of the UI.

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:

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

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

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

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

$$\begin{aligned}&Dc[\![PModel \ \pi_1 \, .. \, \pi_{n_1} \ WidgetName \ \alpha_1 \, .. \, \alpha_{n_2}\\&\quad Category \ \epsilon_1 \, .. \, \epsilon_{n_3} \ Behaviour \beta_1 \, .. \, \beta_{n_4}]\!] =\\&\quad \{\pi_i \mapsto \overline{\pi_i}\}_1^{n_1} \cup \{\alpha_i \mapsto \overline{\alpha_i}\}_1^{n_2} \cup \{\epsilon_i \mapsto \overline{\epsilon_i}\}_1^{n_3} \cup\\&\quad \{\beta_i \mapsto \overline{\beta_i}\}_1^{n_4}\end{aligned}$$

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.

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

We can also define semantic functions which allow us to extract information from the model, for example we define the following which allows us to obtain the set of all behaviours of a presentation model:

$$B[P]$$

where

$$B[P] \mathrel{\widehat{=}} \{[\![P]\!]b | b \in Behaviours(P)\}$$

and $Behaviours(P)$ gives us the identifiers to all behaviours of P (i.e., it is a syntactic operation), and similarly for the other syntactic clauses.

Our presentation models consist of widgets together with names, categories and behaviours. The semantics show 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.

We now give an example where we use the denotational semantics to instantiate the following presentation model:

$$\begin{array}{ll}PModel & p \ q \ r \ s\\Widgetname & w \ v \ y\\Category & c \ d \ e\\Behaviour & b1 \ b2 \ b3\end{array}$$

$$\begin{array}{ll}p \ is & (w, \ d, \ (b2 \ b3))\\ & (v, \ e, \ (b1))\\q \ is & (y, \ d, \ (b2 \ b3))\\r \ is \ p : q & \\s \ is \ r & \end{array}$$

As an abbreviation, we call the declarations (the first four lines above) Dec and the definitions (the last five lines) Dfs in what follows.

$$\begin{aligned}&[\![Dec \ Dfs]\!]\\&=\\&Df[\![Dfs]\!](Dc[\![PModel \ p \ q \ r \ s \ \ WName \ w \ v \ y\\&\quad Category \ c \ d \ e \ \ Behaviour \ b1 \ b2 \ b3]\!])\\&=\\&Df[\![Dfs]\!](\{p \mapsto \overline{p}, q \mapsto \overline{q}, r \mapsto \overline{r}, s \mapsto \overline{s}\} \cup \{w \mapsto \overline{w},\\&v \mapsto \overline{v}, y \mapsto \overline{y}\} \cup \{c \mapsto \overline{c}, d \mapsto \overline{d}, e \mapsto \overline{e}\}) \cup \{b1 \mapsto \overline{b1},\\&\quad b2 \mapsto \overline{b2}, b3 \mapsto \overline{b3}\}\\&=\\&Df[\![Dfs]\!](\{p \mapsto \overline{p}, q \mapsto \overline{q}, r \mapsto \overline{r}, s \mapsto \overline{s}, w \mapsto \overline{w}, v \mapsto \overline{v},\\&\qquad y \mapsto \overline{y}, c \mapsto \overline{c}, d \mapsto \overline{d}, e \mapsto \overline{e},\\&\qquad\quad b1 \mapsto \overline{b1}, b2 \mapsto \overline{b2}, b3 \mapsto \overline{b3}\})\end{aligned}$$

Now we use ϵ in what follows to refer to the populated environment we have given in the last line above. Also, D will abbreviate the definition on the fifth and sixth lines above and Ds the remainder of those definitions.

$$\begin{array}{l}
Df[\![D\ Ds]\!]\epsilon \\
= \\
Df[\![Ds]\!](Df[\![p\ is\ (w, d, (b2\ b3)), (v, e, (b1))]\!]\epsilon) \\
= \\
Df[\![Ds]\!](\epsilon \oplus \{p \mapsto Expr[\![(w, d, (b2\ b3)), \\
\qquad\qquad\qquad (v, e, (b1))]\!]\epsilon\}) \\
= \\
Df[\![Ds]\!](\epsilon \oplus \{p \mapsto Expr[\![(w, d, (b2\ b3))]\!]\epsilon \cup \\
\quad Expr[\![Es]\!]\epsilon\}) \\
= \\
Df[\![Ds]\!](\epsilon \oplus \{p \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\} \cup \\
\quad Expr[\![Es]\!]\epsilon\}) \\
= \\
Df[\![Ds]\!](\epsilon \oplus \{p \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\} \cup \\
\quad Expr[\![(v, e, (b1))]\!]\epsilon\}) \\
= \\
Df[\![Ds]\!](\epsilon \oplus \{p \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\} \cup \\
\quad \{((\overline{v})\ (\overline{e})\ \{(\overline{b1})\})\}\}) \\
= \\
Df[\![Ds]\!](\epsilon \oplus \{p \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), \\
\qquad\qquad\qquad ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\})\}\}) \\
= \\
(letting\ \epsilon' = \epsilon \oplus\ \{p \mapsto ((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), \\
\qquad\qquad\qquad ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\})\} \\
and\ now\ having\ Ds\ abbreviating\ the\ eighth\ and \\
\qquad\qquad ninth\ lines)
\end{array}$$

$$\begin{array}{l}
Df[\![Ds]\!](Df[\![q\ is\ (y, d, (b2\ b3))]\!]\epsilon') \\
= \\
Df[\![Ds]\!](\epsilon' \oplus \{q \mapsto Expr[\![(y, d, (b2\ b3)]\!]\epsilon'\}) \\
= \\
Df[\![Ds]\!](\epsilon' \oplus \{q \mapsto \{((\overline{y})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\}\}) \\
= \\
(letting\ \epsilon'' = \epsilon' \oplus \{q \mapsto \{((\overline{y})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\}\})
\end{array}$$

$$\begin{array}{l}
Df[\![s\ is\ r]\!](Df[\![r\ is\ p\ : q]\!]\epsilon'') \\
= \\
Df[\![s\ is\ r]\!](\epsilon'' \oplus \{r \mapsto Expr[\![p : q]\!]\epsilon''\}) \\
= \\
Df[\![s\ is\ r]\!](\epsilon'' \oplus \{r \mapsto Expr[\![p]\!]\epsilon'' \cup Expr[\![q]\!]\epsilon''\}) \\
= \\
Df[\![s\ is\ r]\!](\epsilon'' \oplus \{r \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), \\
\qquad ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\})\} \cup \{((\overline{y})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\}\}) \\
= \\
Df[\![s\ is\ r]\!](\epsilon'' \oplus \{r \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), \\
\qquad ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\}), ((\overline{y})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\}\}) \\
= \\
(letting\ \epsilon''' = \epsilon'' \oplus \{r \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), \\
\qquad ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\}), ((\overline{y})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\}\})
\end{array}$$

$$\begin{array}{l}
Df[\![s\ is\ r]\!]\epsilon''' \\
= \\
\epsilon''' \oplus \{s \mapsto Expr[\![r]\!]\epsilon'''\} \\
= \\
\epsilon''' \oplus \{s \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\}), \\
\quad ((\overline{y})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\}\}
\end{array}$$

Expansion of ϵ, ϵ', ϵ'' and ϵ''' gives the final environment as:

$$\begin{array}{l}
\{w \mapsto \overline{w}, v \mapsto \overline{v}, y \mapsto \overline{y}, c \mapsto \overline{c}, d \mapsto \overline{d}, e \mapsto \overline{e}, \\
\quad b1 \mapsto \overline{b1}, b2 \mapsto \overline{b2}, b3 \mapsto \overline{b3}, \\
p \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\})\}, \\
q \mapsto \{((\overline{y})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\})\}, \\
r \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\}), ((\overline{y})\ (\overline{d}) \\
\quad \{(\overline{b2})\ (\overline{b3})\})\}, \\
s \mapsto \{((\overline{w})\ (\overline{d})\ \{(\overline{b2})\ (\overline{b3})\}), ((\overline{v})\ (\overline{e})\ \{(\overline{b1})\}), ((\overline{y})\ (\overline{d}) \\
\quad \{(\overline{b2})\ (\overline{b3})\})\}\}
\end{array}$$

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

6 Example

The example we present here is based on a case study which was undertaken to see how the use of our formal models influence a UI design process. It is somewhat self-referential in that the software application being designed is intended to help users create and edit presentation models (and PIMs, which we describe later).

The UI design consists of 26 different screens and dialogue windows which were developed following user requirements analysis. In Fig. 2 we show the prototype for the opening screen of the application. This provides the user with a view of any existing models already stored in the application as well as enabling them to navigate to different parts of the application where they can view or edit these models or add new models.

The presentation model for the prototype given in Fig. 2 is:

PModel
MainWin Widgetname
FileMenu QuitMenuItem EditMenu AddPMMenuItem
AddPIMMenuItem MaxWin MinWin AddPMButt
AddPIMButt PMList PIMList

Category
Container ActCtrl SValSel

Behaviour
QuitApp UI_OpenAddPMDecs UI_OpenAddPIM
UI_MaxWindow UI_MinWindow UI_OpenViewPM

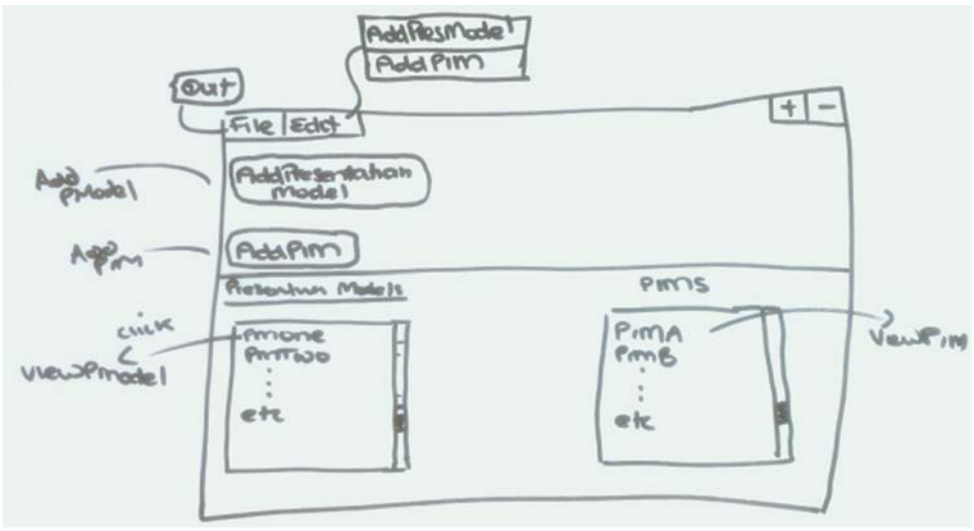

Fig. 2 Prototype for opening screen of PIMed application

$\quad UI\_OpenAddPMDecs\ UI\_OpenAddPIM$
$\quad UI\_OpenViewPM$
$MainWin\ is$
$\quad (FileMenu, Container, ())$
$\quad (QuitMenuItem, ActCtrl, (QuitApp))$
$\quad (EditMenu, Container, ())$
$\quad (AddPMMenuItem, ActCtrl, (UI\_OpenAddPMDecs))$
$\quad (AddPIMMenuItem, ActCtrl, (UI\_OpenAddPIM))$
$\quad (MaxWin, ActCtrl, (UI\_MaxWindow))$
$\quad (MinWin, ActCtrl, (UI\_MinWindow))$
$\quad (AddPMButt, ActCtrl, (UI\_OpenAddPMDecs))$
$\quad (AddPIMButt, ActCtrl, (UI\_OpenAddPIM))$
$\quad (PMList, SValSel, (UI\_OpenViewPM))$
$\quad (PIMList, SValSel, (UI\_OpenViewPIM))$

Each of the widgets in the prototype has been described in terms of its name, category and behaviour. There are two different types of behaviour which a widget can be associated with. An *interaction* behaviour (indicated by a name prefixed with $UI\_$) is any behaviour which affects the UI in some way, either by navigating to a different part of the system (and so presenting a different UI to the user), or by changing some aspect of the current UI (for example by resizing a window). A *system* behaviour is any behaviour which affects the underlying system. We can think of this in terms of a low-level implementation, i.e., as something which causes an event which changes the system state (for example by changing a stored value). We will refer to these as $I\_Behaviours$ and $S\_Behaviours$. (Differentiating between these two types of behaviour proves to be particularly useful when using the models to consider refinement issues of UIs but is not an important distinction within this paper. We will, however, use the distinction for some of the uses we discuss.)

The presentation model does not replace the prototype (i.e., we do not require that our UI designers produce these models instead of prototypes) but rather they act as a bridge between the informal and the formal software development artefacts. We will discuss this further in the following sections.

7 Using the presentation model

7.1 A common design language

One of the problems we identified with separating the UI and system design is that of being sure that all parties have the same end product in mind. It is hard to compare informal design artefacts with a formal specification, for example, and be certain that they mean the same thing (as opposed to having a belief that they do). The presentation model gives us a stepping-stone towards a common language which enables us to gain this certainty. Because the model has a formal structure and a semantics we can be sure of what it means and use this as the basis to start integrating the UI design with the formal process.

As we have already stated, there are a number of different formal methods, languages and notations that exist, all of which can be satisfactorily used to describe systems and prove properties about that description prior to implementation. We intend that the models we develop of the informal artefacts can be used with as many of these different formalisms as possible rather than just tailoring them to one specific style. For the purposes of this paper we will focus on the approach of formally specifying a system in the language Z [1] and then refining that specification (once we have satisfied ourselves it is itself correct and the system it describes is correct also) into an implementation using some stepwise refinement method. We will also make use of the μCharts language [32] which has its semantics given in Z. However, we make this choice from convenience (they are languages we are familiar with) rather than from any intention to tie the use of models to these particular languages and methods.

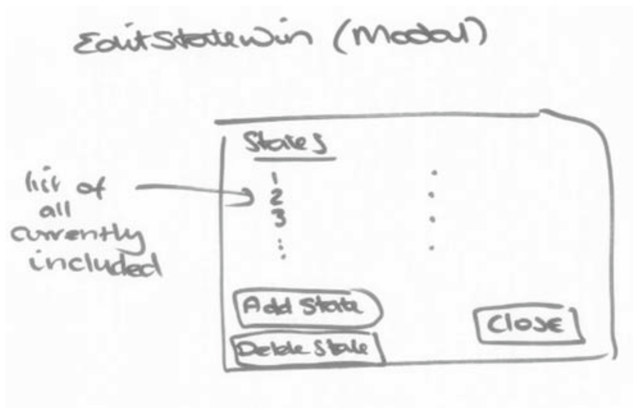

Fig. 3 Prototype for Edit State Window

The first thing we can do with our presentation model is ensure that the behaviour of the UI it describes is the same as that described by the system specification. In Z, the system specification describes an overall state for the system, as well as operations upon that state. If we identify which of those operations relate to actions which a user is required to perform (which can be ascertained from the user requirements) then we can check to see if there are related behaviours in the presentation model. That is, we can create a relation between the operations of the system specification and the behaviours of the UI model. At the very least we expect that every S_Behaviour in the presentation model is related to some operation of the system specification and that any operations identified as user requirements are related to a behaviour in the presentation model.

Returning to our case study example again, we now show a prototype for one of the dialogues, given in Fig. 3. This has the following presentation model (we omit the declarations for brevity):

$EditStateWin\ is$
$\quad (StateList, StatusDisplay, ())$
$\quad (AddStateButt, ActCtrl, (AddNewState))$
$\quad (DeleteStateButt, ActCtrl, (UI\_OpenDelSCheck))$
$\quad (CloseButt, ActCtrl, (UI\_CloseEdSWin))$

One of the behaviours in this model is an S_Behaviour, namely $AddNewState$. We can relate this behaviour to a corresponding operation in the system specification. In this example our system specification includes the following operation schema:

$AddStateToPIMOp$
ΔPIM
$newstate? : PIMSTATE$
$states' = states \frown < newstate? >$

and so we can relate these two. We do the same for each of the S_Behaviours in the model. Once this is complete we expect that every S_Behaviour is related to some operation in the specification, so that they form a subset of the specified operations.

There are usually fewer S_Behaviours than there are specified operations (although this is not necessary) as there will be operations relating to the system which are not directly available to the user (and which we would not want to be available to the user). For example they may be operations relating to security aspects within the system. It is necessary, however, that there are not more S_Behaviours in the presentation model than there are in the specification. If this is the case, it suggests that the UI designer believes that some functionality will be available in the system which is not actually part of the design, i.e., the UI designer has a different understanding of the application being built than the person responsible for the specification.

While we have shown here a simple example where there is a direct correspondence between a single UI behaviour and a single specified operation, frequently behaviours within the UI are more complex and may trigger other behaviours. This requires more work in order to satisfy ourselves that overall behaviour remains correct. We give details of this, and show how we can use a form of data refinement with the presentation model of a UI, in [6].

7.2 Presentation models and design equivalence

Another way in which we can use presentation models is to compare different designs. We do this using different types of design equivalence which are based on properties of 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.

It is not unusual for software to be designed to run on a number of different platforms, some of which may have very different UIs, for example mobile phones or PDAs, where screen size, as well as requirements, affects the design of the UI. Despite the differences in the UIs we want to be sure that they provide the user with the same possibilities to interact, that is we want to be sure that functionally they are the same (even though the way in which the user interacts may be different). We determine the functionality of a design from 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 1 If $DOne$ and $DTwo$ are UI designs and $PM\ One$ and $PMTwo$ are their corresponding presentation models then:

$$DOne \equiv_{func} DTwo =_{df} B[PMOne] = B[PMTwo]$$

As well as functional equivalence we have considered other types of equivalence which exist between designs, namely *component equivalence* and *isomorphism*. These are primarily useful when we are updating an existing application and want to ensure that the UI to the new version of the system is still familiar to the user, as well as correct functionally. 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 properties

The third use of presentation models we present here is their use in ensuring our designs have "desirable properties". Such properties may be defined in a number of different ways. For example they may relate to guidelines describing UI properties or design rules, such as [17] or [35], or they may be based on more general research such as Shneiderman's "Eight golden rules for interface design" [33] or Nielsen's "Top 10 Mistakes in Web Design" [38].

One such property is that of consistency. Shneiderman states [33]:

> 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 a large number of different screens and dialogues, so maintaining consistency throughout is not a trivial task, indeed much research has been undertaken in precisely this area. Researchers such as Thimbleby, for example, have examined this problem from a number of different perspectives including the relationship between modes and consistency [18], use of matrix algebra within UI design [37], etc. Whilst we do not hope to solve the problem of addressing this issue exhaustively using our models, we are able to tackle at least one part of the problem. One of the things we can ensure, using the presentation model, is that controls which have the same behaviour 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 behaviour and this ensures that the user always knows what to expect when they encounter such a control.

Another desirable property we can consider using the presentation model is the reactivity of the interface. The split between I_Behaviours and S_Behaviours within the presentation model, and the categories of the widgets used, give us an indication of how much of the UI is responsive to user interaction, and how much is informative. From this we can determine whether the UI is primarily active or passive.

8 Presentation interaction models

We have provided some examples of how we can use presentation models of informal designs to help with our aim of integration of informal design artefacts into a formal process (via the relation between specification and model), and also in dealing with design concerns such as consistency.

As a first step in considering correctness of the UI design with respect to the specification we can show that the necessary behaviours exist in the UI design. However, it is also necessary to show that the user can actually access all of the functionality described. For example, in our previous example we showed that the specified operation to add a new state had a corresponding behaviour in the dialogue window shown in Fig. 3, but suppose there is no function within the UI which causes this dialogue to be displayed? If that is the case then the user can never access this functionality and our UI design has failed to really satisfy the specified system. We must therefore be sure that not only does all of the relevant functionality exist, but also that it is reachable by the user. To show this we must consider which parts of the UI a user can get to from any other part of the UI.

Proving the reachability properties of a UI is a common concern in much of the early work on using formal methods with UIs. In their early work on PIE models, Dix and Runciman [13] describe *strong reachability* for a UI which states that "not only can you get anywhere but you can get anywhere from anywhere". This is the property we require for our designs and models also. It requires that we understand how the UI changes dynamically from screen to screen as the user interacts. The behaviours which cause these changes are the I_Behaviours of the presentation model. However this does not give us quite enough information to prove reachability.

The problem with trying to capture the idea of dynamic change of the UI via the presentation model is that the model gives us a static view of the design. It shows a complete environment which describes 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 change the model by extending it to include additional information. However, the presentation model does contain enough information for many of the things we wish to use it for and we want to avoid making it so complex that it becomes a burden upon designers or formal practitioners to learn and use. Rather than change it then, we decided to use it in conjunction with another common formalism which would allow us 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 UI modelling in both design (as early as the late 1960's [26]) and as a way of evaluating interfaces [25]. 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 UIs or UI 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 requires only a 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 the following:

- 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
- A start state, q_0, one of the states in Q
- A set of accepting states, F, which is a subset of Q
- A relation, R, which relates states to *PModels*

The FSM is then a six-tuple $(Q, \sum, \delta, q_0, F, R)$.

The relation, R, between presentation model and state of the FSM is used to indicate that 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 for interaction. 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 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 PIM 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,$(Q, \sum, \delta, q_0, F, R)$, we can give this more formally as:

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

where q_{PModel} is the presentation model associated with state q, i.e., $(q_{PModel} \mapsto q) \in R$

Returning again to our case study example, we stated that there were 26 different screens and dialogues which made up the UI. This indicates that there is, therefore, a considerable amount of behaviour within the UI which manages the movement between these different parts. Rather than trying to capture the whole PIM in one view we prefer a modular approach which makes it easier to view and understand, and so we use the μCharts language [32] as a way of visualising PIMs. This allows us to provide both an abstract view of the PIM, such as the top level view we show in Fig. 4, as well as increasingly detailed views, as in the PIM for an individual dialogue which we give in Fig. 5. States represented by ovals are individual states of the PIM, whereas the rectangles indicate a higher level view and can be decomposed into further states (we can consider them as states with a chart embedded within them).

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. For example, suppose we want to prove that the behaviour we were considering earlier, $AddNewState$, is reachable. We know from the presentation model that this behaviour is in the state of the PIM called $EditStateWin$. In Fig. 5 we can see that this state can be reached from the $MainViewPIM$ state via the behaviour $UI\_OpenEdStateWin$, and in Fig. 4 we see that this state is in turn reachable from the start state, $MainWindow$. We can, therefore, be satisfied that this behaviour is reachable. Once we have determined that all of the behaviours are both related to specified operations and are reachable, then we can prove the strong reachability property of the UI by ensuring that the PIM itself has this property. In fact we can prove that the PIM has the strong reachability property first if we choose, but this does not guarantee that all related behaviours are themselves reachable as parts of the presentation model may be missing from the PIM. We cannot discover this without checking individual behaviour.

In addition to properties of the behaviour of the UI, we can also use PIMs to consider desirable properties of the design itself (in the same way that we used the presentation models to examine such properties). The complexity of the PIM gives us some indication as to the complexity of the interface we are describing. For example, if we consider a desirable property of our UI design to be that of ensuring minimum memory load on users (i.e., we want to avoid the need for users to have to remember long sequences of actions to navigate through the UI) we can use the PIM to assist with this. A manual inspection of the graph of a PIM gives an indication of not only the *amount* of navigation required (via the number of states) but also the complexity of navigation between these states, based on graph properties such as cycles.

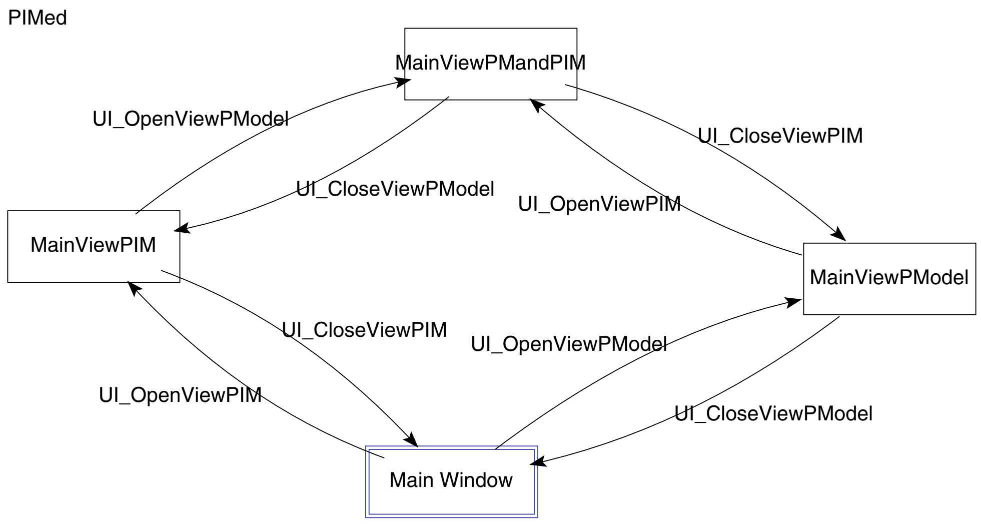

Fig. 4 Top level view of PIM

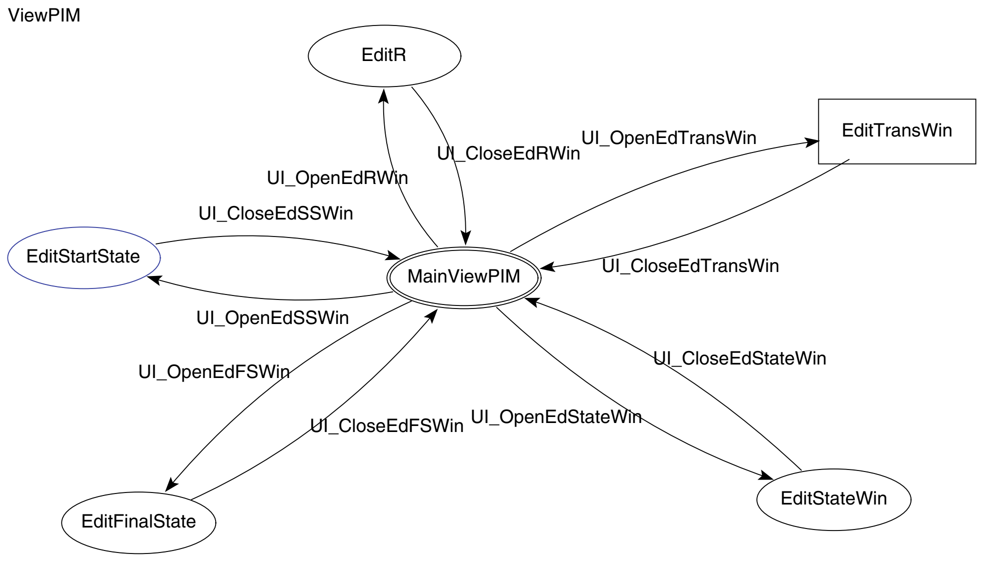

Fig. 5 Detailed view of PIM for ViewPIM window

Consider the PIM given in Fig. 6. The cycle which exists between states A, B and C suggests straightforward navigation between these states (i.e., requiring a single action from the user each time), whereas the chaining between states B, D, E and F indicates increasing complexity of navigation. A user moving from state F back to state A is required to perform four actions, increasing their cognitive load.

Similarly, the larger the number of states of the PIM the more complex we consider the UI (as more navigation is required). If we wish to simplify the design and reduce the number of states we could take the approach of using FSM minimisation techniques as a first step in achieving this, the minimised PIM can be used as a basis for the new design. While minimisation is not enough by itself to perform such a simplification of the UI (we must ensure for example that we do not introduce non-determinism and that the reduced number of states of the UI does not lead to clutter within the different screens) it may prove a useful first step in approaching the simplification problem by ensuring that the basis for the redesign retains the reachability properties already proved.

9 Transforming designs to UIs

Once we have satisfied ourselves that the UI designs are correct with respect to the specification of the system we are building, we still need to ensure that the final implementation of that design is also correct. That is, we need a way to transform our designs into implementations in a way which preserves the correctness. In Sect. 3 we described how the formal process of refinement is used to transform specifications into correct implementations and we now wish to extend this notion for UI designs.

There are two main considerations when approaching refinement for UIs. Firstly, we must define what exactly we mean when we talk about refinement of a UI so that we can

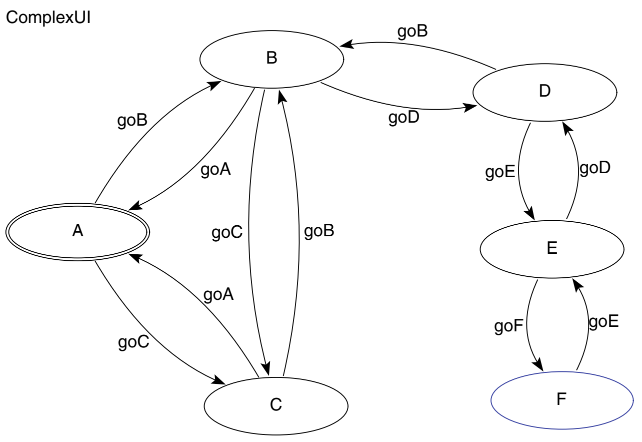

Fig. 6 PIM example with chaining

develop the necessary steps to be followed. Secondly, we need to consider how we put the system and UI together in one formal description. We start to approach both of these questions by using the μcharts language and its refinement theory [32].

9.1 System and UI composition

We describe the formal description of both system and UI together as the composition of the two parts:

$$Sys||UI$$

In order to consider them in this way we need a common language. Currently we have the system described in some specification language and the UI described by way of the presentation models and PIMs of the design. One way to achieve a common language would be to transform the presentation model (or PIM) into the same language as the specification (in our case, Z). However, this would not only require us to model the entire UI state within the Z specification, but also describe the interaction between system and UI via all of the operations. This seems to be a time-consuming and non-intuitive solution as the complexity of the formalism being used is very removed from the visual nature of the design.

Instead, we consider a different approach using the μ Charts language. This language, which began as a simplified version of statecharts, has both a visual representation (which we have already used for visualising PIMs) and an underlying logic and semantics given in Z [19,32]. If we consider the nature of a sequential μchart, it is essentially a finite-state automaton that describes the set of outputs that results from reacting to a set of inputs in a given state. We have already seen that UIs can be considered in this way, and this has allowed the representation of UIs by both FSM [26,39] and statecharts [20] in previous research. We take advantage of the fact that we can consider a UI as a type of reactive system and also that μCharts has a logic and a formally defined refinement theory (which immediately gives us an advantage over a more commonly used language such as statecharts which has neither of these things). In addition μCharts contains a syntax for composition which matches exactly what we want for our $Sys||UI$ description in that it allows communication between μcharts (which are the visual representations of the μCharts language) which can be restricted to prevent the environment (or user) interacting directly with the part of the chart representing the system.

Figure 7 shows a simple example of such a μchart. The chart contains two sequential μcharts in composition with each other. The top chart models the UI design and the bottom chart the system. μcharts react to inputs from the environment (which we consider to be a user) which trigger transitions between states and at the same time signals may be output back to the environment. The charts can also communicate with each other via a mechanism called feedback. The signals which the charts can communicate on are given in the set at the bottom of the chart ($privateI$ and $privateO$ in our example). Whenever one of the charts outputs one of these signals is it instantaneously fed back to the other chart. The signal sets at the left and right hand sides of the chart represent input and output interfaces. These restrict the signals which the chart will accept from the environment and output to the environment. In our example the chart will only accept the signal $userInputs$ from the environment and will output nothing.

The mechanisms we have described for μcharts allows us to model $Sys||UI$ as composed systems which react to some signals from an outside environment and which have a private mode of communication between them. This exactly replicates our view of a system where the user interacts via a UI, rather than with the system directly, and the UI and system communicate privately.

Returning to our earlier example from the case study we could model the $EditState$ part of the system with the μchart of Fig. 8. The user interacts with the top part of the chart via the input signals $Delete$, $DelYes$ and $DelNo$, and the

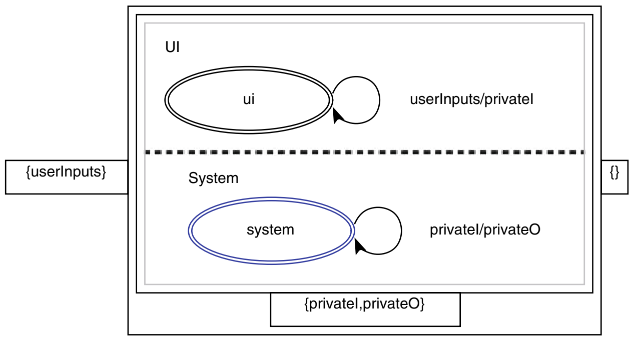

Fig. 7 System || UI

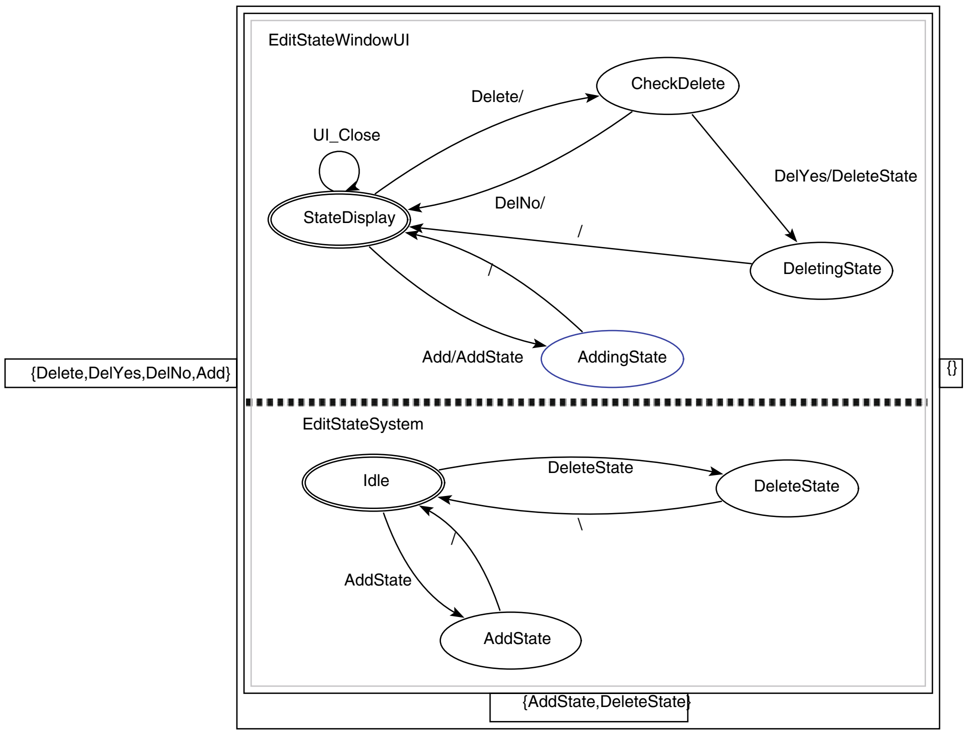

Fig. 8 Composed chart for EditStateWindow

UI and the System interact via the signals $AddState$ and $DeleteState$.

It is not our intention to try and model all $Sys||UI$ pairs for all systems in this manner, rather we are using examples of this approach to generalise the behaviour for the composition so that we can begin to understand the refinement considerations. We will discuss this next.

9.2 Refinement for UIs

In order to formally define what refinement means for UI designs and UIs we must first understand what such a refinement may consist of. Refinement is a formal process of transformation for some design (at any level of abstraction) to something which is less abstract. We can repeat the process until we reach a concrete implementation.

When we consider a UI design, it has both behaviour and appearance. Informal artefacts, such as prototypes, tell us a lot about the appearance of a UI, but as we have discussed, by themselves they do not clarify behaviour. Presentation models and PIMs on the other hand tell us a lot about behaviour, but only give an abstract idea of appearance (by describing the types of widgets that may be used but saying nothing about layout, etc.). We will then need to consider both the formal and the informal when we consider refinement.

We can examine the literature on "traditional refinement" (by which we mean the sorts of formal processes we have discussed in Sect. 3 and which are well documented in such works as [24,40,41]) which gives us some ideas about how we may consider refinement. For example, one general way of characterising refinement (as we have said before) is via the concept of substitution. If we have some system A, and can replace it by system C, and it is not possible to tell that a substitution has occurred then we might state that a refinement has taken place. This turns out to not be an intuitive way to consider UI refinement. We may replicate the functionality of our abstract UI with a concrete UI, but if its appearance is different then any user will recognise that a substitution has taken place. More useful is the notion of contractual obligation. If we have a design which satisfies the user in terms of its behaviour, appearance and usability then any suitable refinement should not reduce satisfaction in any of these areas.

These considerations, and how to formally define such considerations, are the current focus of our research. We aim to use the refinement rules for μCharts to try and characterise UI refinement which will then allow us to investigate the properties of such a refinement, for example in respect of its monotonicity.

10 Conclusion

In this paper we have described two models which allow us to capture the information in informal design artefacts. The models enable us to integrate informal UI design methods with formal methods, as well as providing benefits to the UI design process itself.

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. Our work not only acts as a step towards our goal of integrated formal and informal methods, but also aids in general design issues relating to design of interfaces for different platforms and upgrading of legacy applications.

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.

Having shown how we can integrate early stages of UI design into a formal software development process, we have gone on to describe the next step, that of refining the UI design into an implementation. We have discussed some of the problems this presents, namely finding a way to represent the system and UI together in a common language and defining what the meaning of UI refinement is. We have then shown how we can use the μCharts language as a first step in solving these problems.

An area not covered in this paper, which we believe is worth further study, is that of integrating our models and methods with existing work in the field. In particular our work avoids any attempt to model users or user behaviour. Instead from the point of view of the formal models we treat the user as an abstract entity which has the ability to interact with the system via the specified behaviours; all user concerns and requirements are contained within the UCD process. It would certainly be interesting to extend our work to include issues of user behaviour and cognition. One way to achieve this might be to take an existing approach to user modelling, such as PUM, and find ways to integrate this into our work. Developed initially as an evaluation tool for UIs, PUMs [42] have been subsequently adapted and used extensively by Blandford et al. (see for example [2,3,7]) in approaches to integrating user models with design models and usability evaluation at early stages of the design process. It may be possible to integrate semi-formal PUMs with our presentation models and PIMs in a way that enables us to do things such as examine the correspondence between expected user-behaviours (from the PUM) and behavioural possibilities of the UI (from the presentation model and PIM) leading to a more thorough investigation of early design options.

References

1. 13568 I (2002) Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics, 1st edn. Prentice-Hall International Series in Computer Science, ISO/IEC, Upper Saddle River
2. Blandford AE, Butterworth R, Curzon P (2001) PUMA footprints: linking theory and craftskill in usability evaluation. In: Proceedings of Interact, IOS Press, Amsterdam pp 577–584
3. Blandford A, Butterworth R, Curzon P (2004) Models of interactive systems: a case study on programmable user modelling. Int J Hum-Comput Stud 60(2):149–200. doi:10.1016/j.ijhcs.2003.08.004
4. Bowen J (2005) Formal specification of user interface design guidelines. Master's thesis, University of Waikato
5. Bowen J, Reeves S (2006) Formal models for informal GUI designs. In: 1st International Workshop on Formal Methods for Interactive Systems, Macau SAR China, 31 October 2006, Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam
6. Bowen J, Reeves S (2006) Formal refinement of informal GUI design artefacts. In: Proceedings of the Australian Software Engineering Conference (ASWEC'06), IEEE, pp 221–230
7. Butterworth R, Blandford A (1997) Programmable user models: the story so far, Technical report, Middlesex University
8. Correani F, Mori G, Paternò F (2004) Supporting flexible development of multi-device interfaces. In: EHCI/DS-VIS, pp 346–362
9. Courtney A (2003) Functionally modeled user interfaces. In: Joaquim J, Jardim Nunes N, Falcao e Cunha J (ed) Interactive Systems. Design, Specification, and Verification. 10th International Workshop DSV-IS 2003, Funchal, Madeira Island (Portugal) Springer Verlag Lecture Notes in Computer Science LNCS, pp 107–123
10. Coyette A, Faulkner S, Kolp M, Limbourg Q, Vanderdonckt J (2004) 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, ACM Press, New York, pp 75–82. http://doi.acm.org/10.1145/1045446.1045461
11. Derrick J, Boiten E (2001) Refinement in Z and object-Z: foundations and advanced applications. Formal approaches to computing and information technology, Springer, Berlin. http://www.cs.ukc.ac.uk/pubs/2001/1200
12. Dijkstra EW (1976) A discipline of programming. Prentice Hall, Upper Saddle River
13. Dix A, Runciman C (1985) Abstract models of interactive systems. People and Computers: Designing the Interface, Cook PJ (ed). Cambridge University Press, Cambridge, pp 13–22
14. Dourish P (2006) Implications for design. In: CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems, ACM, New York, pp 541–550. http://doi.acm.org/10.1145/1124772.1124855
15. Duke DJ, Fields B, Harrison MD (1999) A case study in the specification and analysis of design alternatives for a user interface. Formal Asp Comput 11(2):107–131
16. Gieskens DF, Foley JD (1992) Controlling user interface objects through pre- and postconditions. In: Proc. of CHI-92, Monterey, CA, pp 189–194
17. GNOME Human Interface Guidelines(1.0), 2002 (2007) http://developer.gnome.org/projects/gup/hig/1.0/, GNOME Human Interface Guidelines (1.0)
18. Gow J, Thimbleby HW, Cairns PA (2005) Automatic critiques of interface modes. In: DSV-IS, pp 201–212
19. Henson MC, Reeves S (2000) Investigating Z. J Log Comput1 10(1):1–30

20. Horrocks I (1999) Constructing the user interface with statecharts. Addison-Wesley Longman,, Boston
21. Hussey A, MacColl I, Carrington D (2000) Assessing usability from formal user-interface designs. Tech. Rep. TR00-15, Software Verification Research Centre, The University of Queensland
22. IFM07 (2007) http://www.softeng.ox.ac.uk/ifm2007/, http://www.softeng.ox.ac.uk/ifm2007/
23. Landay J (1996) Silk: Sketching interfaces like krazy. In: Human Factors in Computing Systems (Conference Companion), ACM CHI '96, Vancouver, Canada, 13–18 April, pp 398–399, http://citeseer.ifi.unizh.ch/landay96silk.html
24. Morgan C (1998) Programming from specifications, 2nd edn. Prentice Hall, Hertfordshire
25. Paiva A, Tillmann N, Faria J, Vidal R (2005) Modeling and testing hierarchical GUIs. In: Beauquier D, Borger E, Slissenko A (eds) ASM05, Universite de Paris
26. Parnas DL (1969) 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, ACM Press, pp 379–385
27. Paternò F (2001) Task models in interactive software systems. Handbook of software engineering and knowledge engineering
28. Paternò F (2001) Towards a UML for interactive systems. In: EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction, Springer, London, pp 7–18
29. Paternò FM, Sciacchitano M, Lowgren J (1995) A user interface evaluation mapping physical user actions to task-driven formal specification. In: Design, specification and verification of interactive systems, Springer, London, pp 155—173
30. Pfaff GE (1985) User interface management systems. Springer, New York
31. Plimmer B, Apperley M (2002) Computer-aided sketching to capture preliminary design. In: CRPIT '02: Third Australasian conference on User interfaces, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp 9–12
32. Reeve G (2005) A refinement theory for μcharts. PhD thesis, The University of Waikato
33. Shneiderman B (1998) Designing the user interface: strategies for effective human-computer interaction, 3rd edn. Addison Wesley Longman, Boston
34. Smith G (2000) The object-Z specification language. Kluwer, Dordrecht
35. Smith S, Mosier J (1986) Guidelines for designing user interface software. Tech. Rep. ESD-TR-86-278, Mitre Corporation, Bedford, MA
36. Thimbleby H (1990) Design of interactive systems. The Software Engineer's Reference Book
37. Thimbleby H (2004) User interface design with matrix algebra. ACM Trans Comput Hum Interact 11(2):181–236
38. Top 10 Mistakes in Web Design (2007) Jakob Nielsen: Top 10 mistakes in web design, available online. http://www.useit.com/alertbox/9605.html, http://www.useit.com/ Jakob Nielsen's usable information technology website
39. Tsujino Y (2000) A verification method for some GUI dialogue properties. Syst Comput Jpn 31(14):38–46
40. Wirth N (1971) Program development by stepwise refinement. Commun ACM 14(4):221–227, http://www.acm.org/classics/dec95/
41. Woodcock J, Davies J (1996) Using Z: Specification, Refinement and Proof. Prentice Hall, Upper Saddle River
42. Young RM, Green TRG, Simon T (1989) Programmable user models for predictive evaluation of interface designs. SIGCHI Bull 20(SI):15–19. http://doi.acm.org/10.1145/67450.67453