Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 208 (2008) 5–22
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Refinement for User Interface Designs

Judy Bowen[1] and Steve Reeves[2]

Department of Computer Science
University of Waikato
Hamilton, New Zealand

Abstract

Formal approaches to software development require that we correctly describe (or specify) systems in order to prove properties about our proposed solution prior to building it. We must then follow a rigorous process to transform our specification into an implementation to ensure that the properties we have proved are retained. Different transformation, or refinement, methods exist for different formal methods, but they all seek to ensure that we can guide the transformation in a way which preserves the desired properties of the system. Refinement methods also allow us to subsequently compare two systems to see if a refinement relation exists between the two. When we design and build the user interfaces of our systems we are similarly keen to ensure that they have certain properties before we build them. For example, do they satisfy the requirements of the user? Are they designed with known good design principles and usability considerations in mind? Are they correct in terms of the overall system specification? However, when we come to implement our interface designs we do not have a defined process to follow which ensures that we maintain these properties as we transform the design into code. Instead, we rely on our judgement and belief that we are doing the right thing and subsequent user testing to ensure that our final solution remains useable and satisfactory. We suggest an alternative approach, which is to define a refinement process for user interfaces which will allow us to maintain the same rigorous standards we apply to the rest of the system when we implement our user interface designs.

Keywords: Refinement, user interface, formal methods, user-centred design.

1 Introduction

User-centred design (UCD) and an iterative approach to building user interfaces (UIs) allows us to keep users' requirements central to our design and ensure that we consider their feedback as we amend that design. At the same time we can ensure that our interface designs reflect the requirements of both the user and the overall system by incorporating them into a formal design process. We have previously derived a way of integrating UI designs into a formal software development process by way of formal models, [5] and [3], which ensure that the UI and system designers are working towards the same end goal.

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

doi:10.1016/j.entcs.2008.03.104

Having satisfied ourselves that we can consider notions of correctness of a UI design above and beyond the user requirements and design principles, we now turn our attention to the implementation of these UIs. We want to be sure that when our design becomes an implementation we preserve the important properties we have considered during the design stage.

Our system development approach is one where different parts of the system, such as the UI and underlying application logic, are considered separately. This means that as well as relating them during design stages and ensuring that each part is correctly implemented, we must also ensure that the combination of the parts also remains correct.

This then is our motivation for investigating refinement for UIs, to make sure that what we implement is what we intended. We want all of the guarantees of correctness for the UI that we have for the rest of the system. We therefore need some structured and formal way of transforming our designs into implemented UIs, that is we need a refinement process.

One consideration is how we go about generating the code for our UIs. Many software development applications, (such as Visual Studio [12] or Eclipse [7]) utilise 'drag and drop' toolboxes of UI elements which allow quick development of UI layouts and widgets and allow us to delay the programming of behaviour of these widgets. It may be that these UIs are used as interim iterative prototypes and that subsequently we use some other target language for our final implementation, or it may be that we develop these prototypes into fully working UIs and systems within these development environments. In either case there are different stages of development where changes will be made to the UI as we get closer and closer to our end product.

This reflects the incremental approach to system implementation we refer to as *stepwise* refinement [15]. Irrespective of what the intermediate steps are (paper designs, mock-ups, partially functioning UIs, full implementations *etc.*) we want a way of maintaining correctness. This approach to UI refinement is different from that proposed in works such as [6] and [11] in that we are not starting from a single system specification which formalises the UI behaviour as one part of the system, but rather extending traditional UI design methods in a non-traditional, formal manner.

This paper consists of two parts. We start by examining some traditional notions of refinement to see how conceptually they may be applied to UI designs. We will use this as the basis for an informal description of UI refinement and show via some small examples how this may be applied. In the second part we will look at how refinement might be formalised, and introduce the idea of $Sys \;||\; UI$ composition using the μCharts language. We will conclude with a discussion about monotonicity and its importance and future work.

2 Refinement

Refinement is a formal process which allows us to transform one system into another in a manner which ensures that required properties of the original system are preserved. By system we mean any description at any level of abstraction from specification to implementation, or anything in between. Refinement rules can be used to guide the transformation from one system to another, and can also be used to compare two systems to see if one is a correct refinement of another.

Different refinement methods exist for different formal languages, but generally they can be categorised by a common understanding of what the underlying principles of refinement are. We are interested in how these general principles may apply to the concept of UI refinement. That is, how well do they fit with our intuitions about what refining UIs (and UI designs) actually is? We next look at some principles of refinement individually to consider their suitability as principles for UI refinement.

2.1 Principle of Substitutivity

The principle of substitutivity states that it is acceptable to replace one program by another provided it is impossible for a user to observe that the substitution has taken place.

Usually when we talk about substitution we are considering observable behaviours of systems in terms of either input/output traces, or interaction with other parts of the system, *i.e.* behaviour devoid of any notion of visual appearance or cognitive awareness of differences. For UIs, however, such visual and cognitive differences are important; if we substitute one UI for another and they are visually different then the user (who in this case is a real person and not some computer process) will be able to tell that the substitution has taken place. Rather than considering substitutivity we consider the principle behind this concept, namely that of considering programs as contracts.

In [13] Morgan states:

> "A program has two roles: it describes what one person wants, and what another person (or computer) must do."

In this context, refinement must always provide the customer with the ability to do at least the same things they could previously, or more. That is, we can replace one thing with another as long as the customer gets at least what they had before or better (for our purposes by customer we may mean either the end-user or some member of the design team). This is similar to the principle of substitutivity in that it gives conditions under which we can replace one thing with another, but the requirement here is on maintaining utility rather than hiding the substitution. We will refer to this as satisfying *contractual utility*.

As well as the behaviour/functionality of the UI we will also want to consider usability aspects of the UIs, regardless of behaviour; if the replacement UI is perceived to be harder to use than the original then the customer will not be satisfied.

2.2 Decreasing the Level of Abstraction

Through a refinement process our descriptions become less abstract as we add more information, *i.e.* we become more precise about how data is stored or how operations are carried out. This must be done in a manner which avoids inconsistency, so by making more precise decisions about data and operations we must preserve previous correct interactions. The new version should therefore be a specialisation of the previous, more abstract one. Formally, information change must be monotonically increasing (or a least non-decreasing).

One way of adding more information to our UI designs is by defining the categories of the widgets used more precisely. Our formal models for UI designs rely on the widget category hierarchy (originally given in [2]) which enables us to abstractly describe widgets in terms of the type of behaviour they exhibit. An example of the hierarchy tree for *Event Generators* is given in Figure 1.

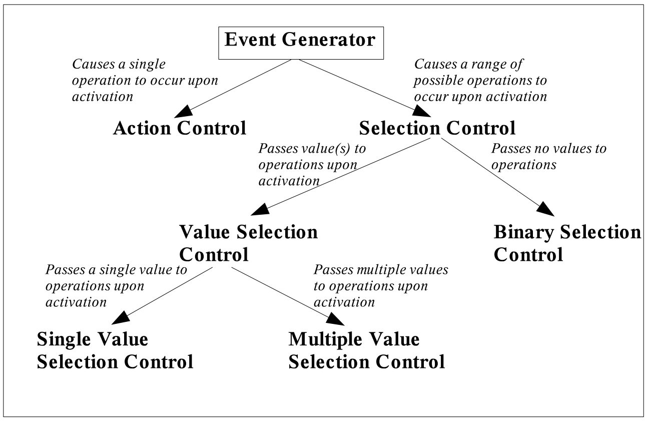
Event Generator
Causes a single operation to occur upon activation
Causes a range of possible operations to occur upon activation
Action Control
Selection Control
Passes value(s) to operations upon activation
Passes no values to operations
Value Selection Control
Binary Selection Control
Passes a single value to operations upon activation
Passes multiple values to operations upon activation
Single Value Selection Control
Multiple Value Selection Control

Fig. 1. Event Generator Hierarchy Tree

We can also use the hierarchy trees to guide a refinement based on this idea of specialisation. For example, we may start off by describing an abstract control, which the user interacts with to choose a value. This is described in the formal model of our early design as an *Event Generator*. At the next step, this could be refined to a *Selection Control*, then subsequently as a *Value Selection Control*, and finally be implemented as a *Single Value Selection Control* (*e.g.* a drop down menu or slider). In this way we already have a process for reducing abstraction by simply following the hierarchy trees.

Another way in which our UIs may become less abstract is in their appearance. We may describe in more detail exactly where the widgets are located and what appearance properties they have (shape, colour *etc.*), so our description becomes more precise.

2.3 Removal of Nondeterminism

We do not generally expect to encounter nondeterminism in UI designs in the same way that we do in system specifications. In system descriptions we ignore nonessential details to postpone decisions about certain behaviours and so nondeterminism arises naturally and is acceptable. The intention of UI designs, however, is to make explicit (so nothing is hidden) to designers and users not only what the UI may look like, but also how it will behave. Nondeterminism arises from deciding to hide information, so if we have nondeterminism we have hidden information, as Hoare [10] states:

> "nondeterminism arises from a deliberate decision to ignore the factors that influence the selection"

If we are hiding information because we have not decided all of the behaviour, then we consider our design incomplete. Whereas there may be parts of the system operations which can remain nondeterministic without affecting our ability to reason about the system this is not true of the UI. Reduction of nondeterminism is not therefore a useful consideration for UI refinement since there should be no nondeterminism to reduce!

3 Refinement and UIs

Having outlined some of the general principles of refinement we now look at how they might apply to UIs. Our UI designs are developed from the requirements of the users, and as such we expect them to include all of the behaviour which has been identified as necessary. We also expect (given we are following a UCD process) that the UIs will be developed following good design principles and with usability for target users in mind.

We have previously described two distinct groups of behaviours of UIs: the *S_Behaviours* which represent the system functionality (where the UI interacts with the underlying system to trigger operations) and the *I_Behaviours* which represent UI functionality, which changes things about the UI itself (for example moving from one part of the UI to another, or changing the size of windows *etc.*) [3], [5]. This UI functionality will not be included in the early requirements as it does not relate to considerations of *what* the system will do, but describes *how* the user will interact with the system and the experience of interacting.

We will consider the *S_Behaviours* and the *I_Behaviours* separately when we begin to define UI refinement, and we will show that there are different requirements for each of them.

Based on the descriptions we have given of the different ways of considering refinement, we state that the following are properties of UIs which we expect to be true for a UI to refine another.

For some arbitrary UIs (or designs) UI_A and UI_C we state that UI_C refines UI_A when:

- we can substitute UI_C for UI_A and maintain contractual utility;
- the widgets of UI_C are not more abstract than those of UI_A;
- the layout and appearance of UI_C is not less defined than that of UI_A;

- the usability of UI_C is not less than that of UI_A.

3.1 Formal Models of UIs

In order to identify the properties of UIs that we are interested in we will use the presentation models and presentation interaction models (PIMs) that represent their designs. The syntax and semantics of these models, along with descriptions of their use, can be found in [5] and [3], however we provide a brief description of them here for clarity.

Presentation models describe a UI in terms of its component widgets. Each widget is described by a triple consisting of:

$$(Name, Category, \{Behaviours\})$$

We distinguish between *S_Behaviours* and *I_Behaviours* by prefixing the behaviour name with an S or I accordingly. The presentation model, therefore, describes the total possible behaviour of a UI (*i.e.* the complete functionality of its implementation).

A PIM on the other hand shows the dynamic behaviour between different states of the UI. It consists of a finite state automaton with a relation between states and component PModels within a presentation model (a PModel is a component description of one part of the UI). When the PIM is in a particular state it indicates that the UI represented by the presentation model related to that state is currently active, and all behaviours of that presentation model are available to a user.

We now have a way of identifying behaviours of the UI and its design formally and a notion of what properties we may wish corresponding UIs to have in order to determine whether or not one refines the other. In the next section we examine each of these properties in more detail and explain how we can identify them using the models.

4 Informally Describing Refinement

4.1 Maintaining Contractual Utility

In order to maintain our contract with the customer the new UI needs to at least provide all of the functionality of the previous UI (and any new functionality has to be consistent with the old). We start by considering the system functionality of the UI, that is the *S_Behaviours*. If we provide a UI which enables a user to interact with the system in n ways, then any replacement UI must at least provide the same n ways of interacting. In fact, we make a stronger statement than that and say that it must provide exactly the same n ways to interact. We will discuss this shortly.

We have previously described different types of equivalence between presentation models which can be used to determine whether two UIs (or designs) are in some way the same [5]. One of these types of equivalence is functional equivalence which has the following definition:

Definition 4.1 If $DOne$ and $DTwo$ are UI designs and $PMOne$ and $PMTwo$ are their corresponding presentation models then:

$$DOne \equiv_{SysFunc} DTwo =_{df} S\_Beh[PMOne] = S\_Beh[PMTwo]$$

Where $S\_Beh[P]$ is a syntactic function that returns the identifiers for all of the $S\_Behaviours$ in P.

We rely on the relation between the identifiers of behaviours and system operations to ensure that those behaviours with the same identifier have the same actual behaviour.

We use this to describe the requirement on $S\_Behaviours$ that we consider is needed to maintain customer satisfaction and state that as long as $\mathrm{UI}_C \equiv_{SysFunc} \mathrm{UI}_A$ then contractual utility is maintained.

It may seem unusual to describe refinement in terms of equivalence in this way, however this is because of the nature of interaction between user, UI and system. We are not considering the *total* functionality of the system here, just the system functionality given in the presentation model (by $S\_Behaviours$), which is the system functionality made available via the UI.

It still appears that this requirement of equivalence is too strict. What if UI_A provides functionality a, b and c to the user and replacement UI_C provides a, b, c and d? We might think that this maintains contractual utility as the user can still do everything they could previously, and in fact they are provided with an added benefit as they can now also do d. However, we need to remember that we are considering the UI in isolation from the underlying system. If we add some widget to the UI intended to perform behaviour d, we have no guarantee that the underlying system actually supports this behaviour. We may end up promising something to the user by providing a widget which does not actually do what we intended. In this case the user will certainly not be satisfied. It turns out that this strictness subsequently restricts the set of valid refinements we allow for UIs but is necessary to guarantee correctness.

We now turn our attention to the UI functionality, or $I\_Behaviours$. Again, the user will expect to be able to do at least as much as they could before. However, because UI functional requirements are not described fully prior to design stages (as we have explained they are not part of user requirements necessarily but a function of the UI itself) it *is* acceptable for these to increase. In this case if we add new $I\_Behaviours$ we do not run the risk of these being unsupported by the underlying system as they relate only to the UI. We state that our requirement for $I\_Behaviours$ is:

$$I\_Beh[UI_A] \subseteq I\_Beh[UI_C]$$

Where $I\_Beh[P]$ is a syntactic function that returns the identifiers for all of the $I\_Behaviours$ in P (where, as before, if identifiers are the same then intended actual behaviour is likewise the same).

As an example of these considerations we present two UI designs and their presentation models. The UIs are for an application which allows a user to display two different shapes. UI_A, on the left of Figure 2 is the original design and UI_C, on the right of Figure 2 a suggested refinement. The presentation models for the designs are:

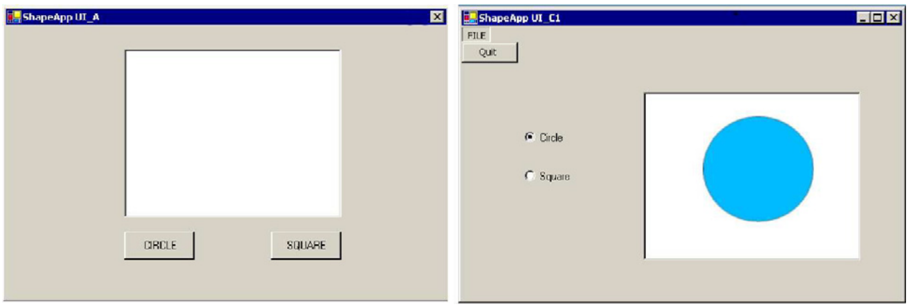

Fig. 2. Design UI$_A$ and UI$_C$

UI$_A$ is (CircleButt, ActCtrl, (S_ShowCircle)),
(SquareButt, ActCtrl, (S_ShowSquare)),
(ShapeFrame, SValRspndr, (S_DisplayShape)),
(QuitButt, ActCtrl, (I_QuitApp))

UI$_{C1}$ is (CircleRB, RadioButton, (S_ShowCircle)),
(SquareRB, RadioButton, (S_ShowSquare)),
(ShapeFrame, SValRspndr, (S_DisplayShape)),
(FileMenu, Container, ()),
(QuitMenuItem, ActCtrl, (I_QuitApp)),
(QuitBox, ActCtrl, (I_QuitApp)),
(MinBox, ActCtrl, (I_MinWindow)),
(MaxBox, ActCtrl, (I_MaxWindow))

From the presentation models we can derive the following:

$$S\_Beh[UI_A] = \{S\_ShowCircle, S\_ShowSquare, S\_DisplayShape\}$$
$$I\_Beh[UI_A] = \{I\_QuitApp\}$$
$$S\_Beh[UI_C1] = \{S\_ShowCircle, S\_ShowSquare, S\_DisplayShape\}$$
$$I\_Beh[UI_C1] = \{I\_QuitApp, I\_MinWindow, I\_MaxWindow\}$$

Comparing these sets shows us that:

$$UI_A \equiv_{SysFunc} UI_C1$$
$$I\_Beh[UI_A] \subseteq I\_Beh[UI_C1]$$

That is, UI_C meets the requirements we have described for maintaining contractual utility and in that respect might be considered a correct refinement of UI_A.

4.2 Less Abstract Widgets

For widget abstraction we are not concerned with behavioural properties, but rather the category of, or actual, widget used. We have given an example of one part of the widget category hierarchy in Figure 1. Similar hierarchy trees exist for *EventResponders* and *Displays*.

For a widget description, becoming less abstract means moving down the relevant hierarchy tree from the current position. As long as the previous category given is a parent node of the new category then we have correctly refined that widget. It is also acceptable for the widget category to remain unchanged (as we may already be at a leaf node describing a particular widget or may have refined some other part of the UI and left some widgets unchanged.) We need only ensure that if a widget category *has* changed that we have not become more abstract (*i.e.* moved up the tree) or that we have selected a widget category which is not a child of the previous one, *i.e.* become incorrectly less abstract.

In the example given in Figure 2, UI_A has standard buttons whereas UI_{C1} has radio buttons. As both of these are examples of *ActionControls* this is a satisfactory refinement. If, however, we were to produce a design which uses a slider to control the chosen shape then we would say that this is not a satisfactory refinement as a slider is an instance of a *SingleValueSelector*, which is not a child of *ActionControl*.

This is an example of using the hierarchy and refinement to support design guidelines by avoiding inappropriate use of widgets. The GNOME Human Interface Guidelines [8] for example, describe the correct use for a slider as:

> "... to quickly select a value from a fixed, ordered range, or to increase or decrease the current value."

This is not the intention of the control as used to select discrete shapes. Using the hierarchy trees to support refinement allows us to avoid such incorrect usage without the need to refer to the guidelines, and additionally may support more inexperienced designers in this area.

4.3 More Defined Appearance

This concept relates to the position and style of the widgets as well as the overall layout appearance (such as background colours, window size *etc.*). These are the low-level details of the UI which are not included in the presentation model and so we cannot use these, or PIMs, to check that a UI is more defined than some other UI. However, in cases where the *only* refinement said to have taken place is that of defining appearance, we can check, via the presentation model and PIM, that this is really the case.

For example, if we have reached a satisfactory final design (perhaps using a support tool such as Visual Basic) and wish to then implement it in some other target language we may expect some of the visual details to change, but not the behaviour. For small examples we may be able to check this by inspection, but for any non-trivial UI we can do this by ensuring that the two UIs are functionally equivalent, *i.e.* the sets of all behaviours in the presentation models are the same, and therefore ensure our final UI correctly implements our ealier design.

4.4 Maintaining Usability

Although presentation models and PIMs were originally developed with the intention of incorporating UI designs into a formal software development process, they can also be used to check for desirable design properties of UIs relating to usability concerns (some examples of this are given in [4]). These are the same sorts of properties we are interested in when we talk about maintaining usability. In order to ensure that a user's experience of using the UI does not get worse we need to make sure that the level of usability we had in our earlier UI (as defined by the desirable properties) is the same, or better, in the new UI. That is, we should not introduce any usability problems where they did not exist before. This does not, of course, take into account the idea of subjective satisfaction. It may be that a user prefers the previous UI because of familiarity, aesthetics, or some other reason. We are concerned here only with impersonal, measurable usability concerns.

One way in which we can test for maintenance of usability is by examining some of the conditions we can test for using PIMs, such as reachability and lack of deadlock. If we have a UI which produces a PIM with strong reachability (by which we mean any state can be reached from any other state) and no deadlock, then we expect that these properties will be preserved in the PIM of the new UI, or we say that usability has not been maintained.

To demonstrate this consider another possible UI for the shape application which we give in Figure 3. The presentation model for UI$_{C5}$ is:

```
UIC5 is MainWin : SquareWin : CircleWin
MainWin is     (SquareCtrl, ActCtrl (I_OpenSqrWin, S_ShowSquare))
               (CircleCtrl, ActCtrl, (I_OpenCrcWin, S_ShowCircle))
               (MinCtrl, ActCtrl, (I_MinWindow))
               (MaxCtrl, ActCtrl, (I_MaxWindow))
               (FileMenu, Container, ())
               (QuitMI, ActCtrl, (I_QuitApp))
SquareWin is   (ShapeFrame, SValResponder, (S_ShowSquare))
               (CloseBox, ActCtrl, (I_OpenMainWin))
CircleWin is   (ShapeFrame, SValResponder, (S_ShowCircle))
               (QuitButt, ActCtrl, (I_QuitApp))
```

The PIMs for both the original design, UI$_A$ from Figure 2, and UI$_C$5, are given in Figure4. The PIM for UI$_A$ is straightforward as there is only one PModel in the presentation model, so the relation R is simply:

$$1 \mapsto UIA$$

1 is both the start state and final state. The PIM for UI$_{C5}$ has three PModels,

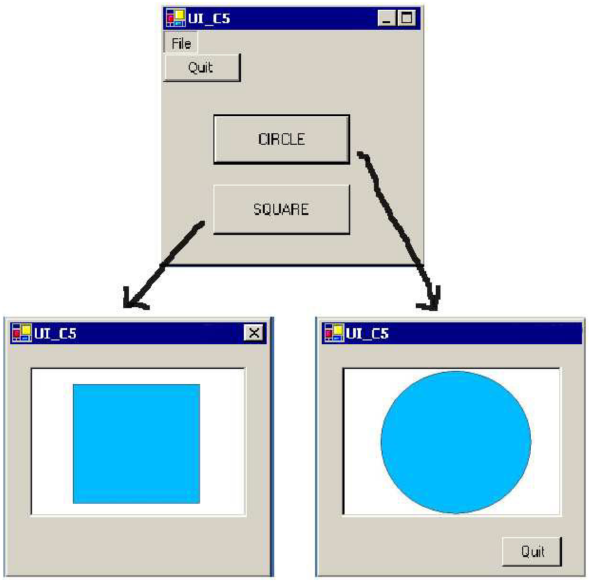

Fig. 3. Design UI_{C5}

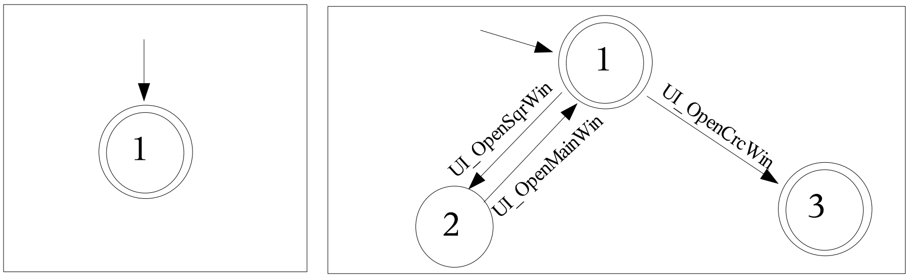

Fig. 4. PIMs for UI_A and UI_{C5}

leading to a relation R which is:

$$\begin{aligned} &1 \mapsto \mathit{MainWin}\\ &2 \mapsto \mathit{SquareWin}\\ &3 \mapsto \mathit{CircleWin} \end{aligned}$$

1 is the start state, and both 1 and 3 are final states.

The PIM for UI_A consists of a single state and so we can be immediately satisfied that it has strong reachability and no deadlock. If we look at the PIM for UI_{C5} we can see that we maintain the deadlock-free state (as we can always reach a final state), but we no longer have strong reachability. It is not possible to reach state 2 from state 3. We cannot therefore say that the new UI maintains usability as it has more restrictions on the availability of behaviours than the previous UI. This breaks our requirement and so we state that UI_{C5} does not maintain the usability of UI_A and is, therefore, not a suitable refinement.

5 Summary of Informal Refinement Description

We have shown how we can use standard refinement concepts, such as contractual utility and reduction of abstraction, to consider refinement of UIs and designs. We have also shown how we can use presentation models and PIMs to examine some of the properties of UIs which relate to these concepts. In order to move on and formalise refinement for UIs we must consider the following: What can we formalise? How can we formalise it? What will we achieve by this?

There are some things we have identified as being desirable for UI refinement which relate to parts of the UI not covered by the models. For example, visual aspects of making appearance less defined (by deciding on colour schemes, appearance styles *etc.*) cannot be checked using presentation models or PIMs. The formal models are concerned with behavioural aspects of UIs, types of widgets of UIs (by which we mean their category) and dynamic movement within the UI, which determines availability of behaviour. If we wish to create a formal definition of UI refinement based on our existing models we must accept that there will be some limitations.

We could argue that those things which we cannot test for are not important considerations, and that the functionality of a UI is the same regardless of whether its background is blue or yellow for example. However, we are mindful that usability considerations *are* important, and these are things which may be affected by aesthetic decisions. We accept that these remain outside of our work and maintain our belief that our methods should be used in conjunction with more traditional design methods, which includes usability testing designed to ensure we do not reduce usability.

We will then formalise the properties which are captured by the models, namely maintaining contractual utility by ensuring equivalence of S_Behaviours and a subset relation for *I_Behaviours*; reducing abstraction by correctly refining widgets based on the category hierarchy trees; maintaining usability by ensuring we do not increase or introduce deadlock or reduce reachability.

For the remainder of this paper we will focus on the first of these, maintaining contractual utility, and discuss ways of formalising this.

6 Formalising Maintenance of Contractual Utility

We must first decide which language or notation to use to describe our refinement rules. In our earlier work on presentation models [5], we showed how they can be integrated with system specifications using the Z language [1]. We reiterate here that we do not want to try and describe our UIs as part of a Z specification (for the reasons we have outlined concerning how UIs are developed), but we recognise that it may be a useful language to help provide a basis for our refinement theory. However, rather than trying to use Z to describe the desired properties of the formal models, we will use a related language which has several advantages over Z, most notably a visual appearance which is more intuitively acceptable as a notation for

UIs.

The language we will use is μCharts.[3] μCharts is a visual, Statechart-like language used for describing reactive systems, which has both a logic and a refinement theory [9], [14]. The language is visually represented by μcharts which are modular in nature, that is, they can be composed together or embedded within states of some other μchart. This ability to compose charts together and the existence of a monotonic refinement theory for such composed charts is one of our reasons for choosing the language.

In general we will not want to model entire UIs as μcharts, as not only does this lead to complex and unwieldy visual representations due to the amount of detail of the UI, but it goes against our desire to keep the formal models we use as simple as possible and as closely related to the designs we are dealing with. What we can do, however, is model the PIMs of our UIs as μcharts (it can be shown via simple examples of total UI models in μCharts that this abstraction is in fact the same thing). That is, the total behaviour exhibited in a full UI model is the same as that exhibited by a correctly described PIM.

By modelling a UI and system pair as a composed μchart we can not only check our informal requirements of functionality equivalence and subset inclusion, but we can also examine how this relates to the monotonic refinement rules for μCharts and what else this may tell us about UI refinement in particular and refinement for interacting systems in general.

7 Composing the UI and System

Returning to our earlier example of the simple shape application, we now show how we would model this along with related parts of the underlying system as a composed μchart. In Figure 2 we presented a UI design for a simple shape application, *UIA*. We now give the composed μchart for the PIM of this UI, along with the underlying system, in Figure 5.

Because we want the user to interact only with the UI, and not directly with the underlying system we constrain the external signals which are visible to the chart using input and output interfaces. The set of signals given in the rectangle at the left hand side of the chart, $\{SShowCircle, SShowSquare\}$, represent the input interface to the chart. Only signals in this set will be accepted from the environment (by which we mean from outside of the chart and for our considerations we can imagine this to be the user) and responded to by the chart. The set of signals given in the rectangle at the right hand side of the chart (which in this example is empty) represents the output interface. Only signals in this set will be visible outside of the chart. The rectangle at the bottom of the chart contains the set of signals which the two parts of the composition can use to communicate with, {DrawCircle, DrawSquare}.

The behaviour of the described system then is that it starts in the states repre-

[3] With a capital ‘C’ it is the name of the language whose primary objects are μcharts (lowercase ‘c’)

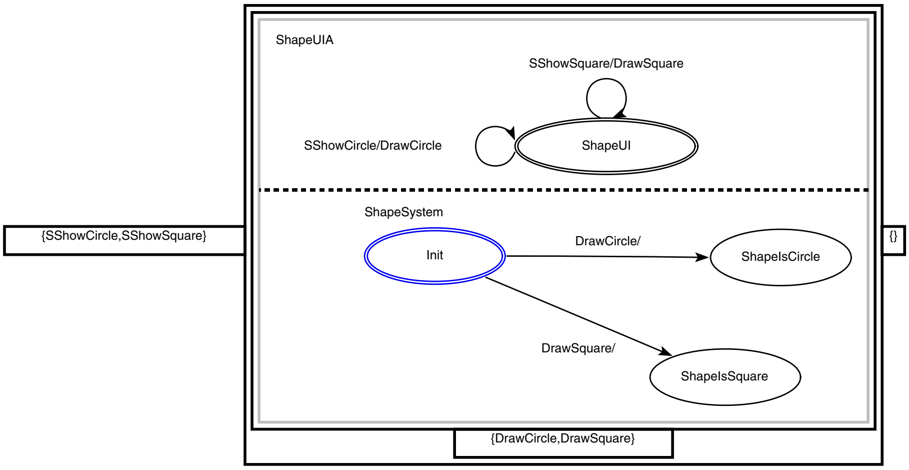

Fig. 5. Composed Chart for Shape Application

sented by a double-lined oval, that is the top chart starts in the *ShapeUI* state and the bottom chart starts in the *Init* state. If the signal *SShowCircle* is seen on the input, then the top chart makes a loop transition and remains in the *ShapeUI* state and outputs the signal *DrawCircle*. Feedback in μCharts is instantaneous, so at the same time the bottom chart sees the *DrawCircle* signal and makes a transition to the *ShapeIsCircle* state. Similarly, if the *SShowSquare* signal is input when the charts are in their initial states then the transitions to *ShapeUI* and *ShapeIsSquare* are made. For this example we assume the *do nothing* semantics of μCharts where nothing happens if a signal appears for which there is no defined behaviour.

The meaning of a μchart is given in the underlying logic by way of a transition model, which we simplify here as being the disjunction of all possible transitions of the chart (including a special *do nothing* transition). In [14] Reeve presents two alternative views of refinement, one based on traces and an equivalent notion based on partial relation semantics. For simplicity and brevity we will talk about trace refinement in this paper.

8 Trace Refinement

When we talk about the traces of a μchart we mean the sequences of input and output sets of signals that model the behaviour of the described system. It is an abstraction of the state-based view and considers only the interactions of the charts and as such it fits neatly with our PIM description of the UI which is a similar abstraction. As an example, consider again the chart given in Figure 5. One possible pair of sequence of traces for this chart is:

$$(\langle\{SShowCircle\}\rangle,\ \langle\{\}\rangle)$$

another possibility is:

$$(\langle\{SShowSquare\}\rangle,\ \langle\{\}\rangle)$$

where we just happen to have singleton traces. In fact, because we are following the *do nothing* semantics for μCharts we can say that there are exactly two possible traces which are:

$$(\langle\{\mathit{SShowCircle}\},\ \epsilon\rangle,\ \langle\{\},\ \epsilon_0\rangle)$$
$$(\langle\{\mathit{SShowSquare}\},\ \epsilon\rangle,\ \langle\{\},\ \epsilon_0\rangle)$$

where ϵ represents any sequence of input sets with elements drawn from *SShowCircle* and *SShowSquare* and ϵ_0 represents the corresponding sequence of empty output sets.

There are two distinct types of refinement in μCharts. The first is *behavioural* refinement, where we remove nondeterminism from a chart by redefining its behaviour, the second is *interface* refinement where we change the input and output interfaces. We have already commented on nondeterminism and so it may appear that *behavioural* refinement is not an important part of our considerations, however when we come to consider the composed chart as a whole (where one part of the composition represents the underlying system which may be nondeterministic) we cannot ignore it completely. Of more interest for the UI part of the composition, however, is interface refinement. Changing the input and output interfaces changes the ways in which the environment can interact with the chart, which for us means changing the ways a user interacts with the UI. We will show that this in fact gives us exactly the same restrictions we have described on how we can change *I_Behaviours* and *S_Behaviours*.

The definitions for input refinement ($\approx_I$) and output refinement ($\approx_O$) are:
For arbitrary charts A and C

$$\begin{aligned} C \approx_I A =_{def} & \ \forall\, i;\ o \bullet (i_{\rhd(in_C)}, o) \in [\![C]\!] \Leftrightarrow (i_{\rhd(in_A)}, o) \in [\![A]\!] \\ & \qquad \wedge\ out_C = out_A \\ C \approx_O A =_{def} & \ \forall\, i;\ o \bullet (i, o_{\rhd(out_C)}) \in [\![C]\!] \Leftrightarrow (i, o_{\rhd(out_A)}) \in [\![A]\!] \\ & \qquad \wedge\ in_C = in_A \end{aligned}$$

where $i_{\rhd in_x}$ restricts the range of the sequence i to the signals in the set in_x and similarly for $o_{\rhd out_x}$. So, informally we can say that interface refinement holds when all (restricted) sequences of traces of the refined chart are traces of the original chart. For simplicity we can consider $[\![A]\!]$ as the set of all traces of the chart A.

8.1 Example

Returning again to our simple shape application and the possible UI designs for that system we will give an example of using trace refinement. In Figure 2 we gave two possible UI designs for the shape application, *UIA* and *UIC*1. In Figure 6 we give the μcharts for the PIMs of these designs.

Note that there are no explicit interfaces defined for these charts, this is a syntactic shorthand for a chart where *every* signal is in the interface, that is there is no filtering or restriction taking place. So the respective interfaces for these two charts

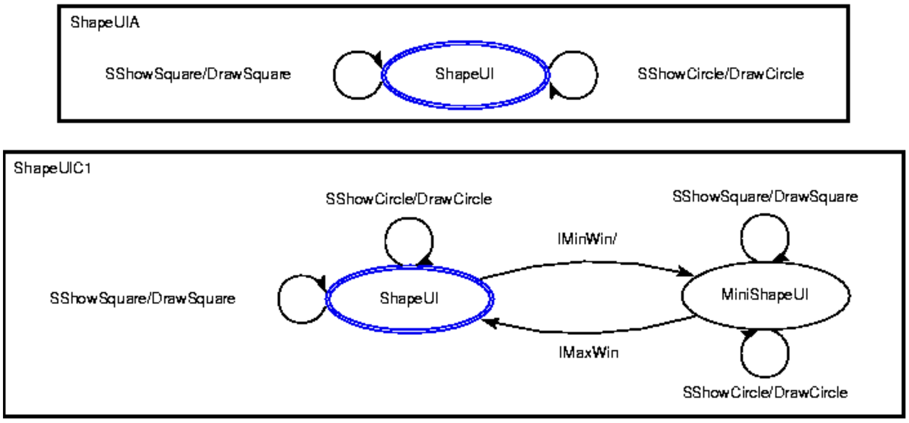

Fig. 6. Sequential Charts for UIA and UIC1

are:

$$\begin{aligned} in_A &= out_A = \{SShowCircle, SShowSquare, DrawCircle, DrawSquare\} \\ in_{C1} &= out_{C1} = \{SShowCircle, SShowSquare, DrawCircle, DrawSquare, \\ &\qquad IMinWin, IMaxWin\} \end{aligned}$$

In our earlier discussions on contractual utility we had stated that $UIC1$ was an acceptable replacement for UIA as it had equivalent system functionality and the UI functionality of the original design was a subset of that of the new design. We now consider the traces to see if we can likewise deduce a refinement. At first glance it appears that there will be a problem with this refinement as there is a trace of $ShapeUIC1$ which is not a trace of $ShapeUIA$, namely $(\langle\{IMinWin\}\rangle, \langle\{\}\rangle)$. However, recall that it is the restricted traces we are interested in, therefore for a trace (i, o) which is $(\langle\{IMinWin\}\rangle, \langle\{\}\rangle)$ we need to consider if $(i \rhd_{in_{ShapeUIA}}, o)$ $\in [\![ShapeUIA]\!]$. If we apply the restriction we are left with the trace $(\langle\{\}\rangle, \langle\{\}\rangle)$ which is a trace of the chart $ShapeUIA$. Similarly we can show that all (restricted) finite sequences of i's and o's which are traces of ShapeUIC1 are traces of ShapeUIA which satisfies the requirement for $\approx_I$ (proof of this is beyond the scope of this paper but we hope that the charts themselves are simple enough that the reader may satisfy themself that this is true.)

9 Monotonicity

We stated earlier that the nature of trace refinement for μCharts reflected the strictness on the conditions we had placed on contractual utility with respect to $S\_Behaviours$. So far we have looked at an example of input interface refinement where no such conditions are necessary, however if we now return to the composed μchart we presented in Figure 5 we can start to understand why this is true.

One of our reasons for looking at UI refinement in terms of μChart refinement was an attempt to define UI refinement formally in a way which would also provide a monotonic refinement with respect to the composition of UI and underlying system. By this we mean that any refinement of one part of the composition implies a refinement of the specification as a whole.

In his description of refinement for μCharts [14], Reeve shows that the composition of μcharts is monotonic with respect to refinement, but requires certain side-conditions to hold. Of most interest here is the second side condition given, which states that for arbitrary composed charts $A = (A1 \mid\mid B)$ and $C = (C2 \mid\mid B)$ it is necessary that $out_{A1} \cap \Psi = out_{C2} \cap \Psi$ where Ψ is the feedback set the charts communicate on. If we were permitted to add or remove *S_Behaviours* from our UI then we would be unable to meet this side condition.

Returning once more to our example in Figure 5, removal of any *S_Behaviours* (which are represented in the composed chart by output signals from the UI chart which are in the feedback set) will change the feedback set and hence change the intersection between feedback and outputs of *ShapeUIA*. Similarly if we were to increase *S_Behaviours* (for example we might add a transition to *ShapeUIA* which responds to the input signal *SShowTriangle* and outputs *DrawTriangle*) it would increase the feedback set and therefore also change the intersection. Of course, we could always omit the new signal from either the feedback set or the output of the chart which would ensure that the intersection between outputs and feedback remained the same, but this would give a nonsensical specification which exactly highlights the problem of adding *S_Behaviours* to UIs where there is no corresponding system operation.

10 Conclusions

In this paper we have discussed the idea of refinement for UIs and shown how we can develop an informal view of this based on traditional notions of refinement. We have then shown how we can use a language such as μCharts to begin to capture this formally. This work should not be seen as yet another attempt to apply some formal method or model to UI design, but rather another step in our aim of incorporating real-world UI design techniques into a formal software development process.

We have explained why we need such a refinement description for UIs, in order to ensure that the properties and guarantees we have made about the early designs are maintained when we implement those designs. We have also described a number of different views of refinement based on traditional notions for system refinement and shown how these may be applied to UIs. This has enabled us to develop an informal notion of refinement which links these traditional views with the characteristics of UIs we consider important and which we can capture using presentation models and PIMs of UI designs. We have briefly discussed the idea of describing systems and UIs in composition and introduced a way of doing this using the language μCharts. Finally we have taken one view of refinement for μCharts, trace refinement, and shown how this can capture one part of our consideration of refinement for UIs, namely contractual utility.

This work gives us a foundation to move forward and complete the formal description of UI refinement within the context of the μChart language. The next step is lift this refinement back into the partial relation semantics of μCharts and extend the parts of UI refinement which we are considering. We also need to examine all of the side conditions required for monotonicity of refinement for composed μcharts and consider their implications on our work. We keep in mind our original aim, which was to find a way of allowing UI designers to develop and design UIs in ways which are practical and intuitive (keeping in mind good design and usability concerns) and at the same time enable formal practitioners to include such designs into a formal software development process consisting of specification, verification and refinement. We believe the work described in this paper is another step forward for this aim.

References

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

[2] Bowen, J., "Formal Specification of User Interface Design Guidelines," Masters thesis, Computer Science Department, University of Waikato (2005).

[3] Bowen, J. and S. Reeves, *Formal models for informal GUI designs*, in: *1st International Workshop on Formal Methods for Interactive Systems, Macau SAR China, 31 October 2006* (2006).

[4] Bowen, J. and S. Reeves, *Using formal models to design user interfaces, a case study*, in: *HCI 2007: Proceedings of the 21st BCS HCI Group Conference*, 2007.

[5] Bowen, J. A. and S. Reeves, *Formal refinement of informal GUI design artefacts.*, in: *Proceedings of the Australian Software Engineering Conference (ASWEC'06)* (2006), pp. 221–230.

[6] Duke, D. J. and M. D. Harrison, *Mapping user requirements to implementations*, Software Engineering Journal **1** (1995), pp. 13–20.

[7] *Eclipse - an open development platform*, Homepage for the Eclipse development organisation. URL http://www.eclipse.org

[8] GNOME Human Interface Guidelines(1.0), 2002 http://developer.gnome.org/projects/gup/hig/1.0/.

[9] Goldson, G., G. Reeve and S. Reeves, *μ-Chart-based specification and refinement*, in: *Formal Methods and Software Engineering. 4th International Conference on Formal Engineering Methods, ICFEM 2002*, LNCS 2495 (2002), pp. 323–334.

[10] Hoare, C. A. R., *Communicating sequential processes*, Commun. ACM **26** (1983), pp. 100–106.

[11] Hussey, A., 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).

[12] *Microsoft visual studio*, microsoft technical pages for the Visual Studio Software. URL http://msdn2.microsoft.com/en-us/vstudio/default.aspx

[13] Morgan, C., "Programming from specifications (2nd ed.)," Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1998.

[14] Reeve, G., "A Refinement Theory for μCharts," Ph.D. thesis, The University of Waikato (2005).

[15] Wirth, N., *Program development by stepwise refinement*, Communications of the ACM **14** (1971), pp. 221–227.