DOI 10.1007/s00165-008-0095-2
BCS © 2008
Formal Aspects of Computing (2009) 21: 589–612

Formal Aspects of Computing

Refinement for user interface designs

Judy Bowen and Steve Reeves

Department of Computer Science, University of Waikato, Private Bag 3105,
Hamilton 3240, New Zealand. E-mail: jab34@cs.waikato.ac.nz

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 both their initial needs and requirements as well as their feedback during the design process and 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, [BR06a] and [BR06b], which ensure that the UI and system designers are working towards the same end goal. Such models also allow us to prove properties about the design itself, including consideration of usability properties [BR07a].

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.

Correspondence and offprint requests to: J. Bowen, E-mail: jab34@cs.waikato.ac.nz

Our system development approach is one where different parts of the system, such as the UI and underlying application logic, are initially 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 [Mic07] for example) 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 [Wir71]. 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 [DH95] and [HMC00] 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.

In [BR07c] we introduced this idea of refinement for user interfaces by first considering what this means conceptually, and giving an informal description of what refinement for UIs might be. We also discussed how we might begin to formalise this description using the μCharts language[1] and how we can create a model of the system and UI together using μCharts. In this paper we expand on this by outlining some of the implications of composing systems and UIs in this way and introduce the concept of *validity* of applications which places restrictions both on the systems and UIs which can be composed, and on the way the compositions are modelled as μcharts. This allows us to discuss the notion of monotonicity of our refinement and explain the correspondence between the validity requirements for system and UI pairs and monotonicity requirements within μCharts. We also look at the relationship between a formal process of UI refinement and existing methods used by designers to transform their designs into implementations.

In considering refinement as a transformation from a description of a system to some other description (generally an abstract description, such as a specification, to a concrete description, such as an implementation) we also have the possibility of considering refinement between implementations. That is, we can compare different implemented versions of the same system (perhaps designed to run on different platforms) and consider whether they are really implementations of the same system. That is, we can discover if there is a refinement relation between them. We will make the distinction between this sort of refinement, which we call vertical refinement, and refining a specification to an implementation, which we call horizontal refinement.

2. Related work

The ideas behind UI refinement have been previously explored in a number of different ways. One recent approach has been within the work of groups involved in the XIML [PE02], [XIM] and UsiXML [LVM+04], [USI] projects. Both of these use Xml-based models of UIs and have a variety of supporting tools, such as Paternò's TERESA [PS02], [PMS03] for example, which enable transformation of models into concrete UIs or provide a mechanism for translating a model into a number of UIs for different platforms or contexts of use. In both of these cases the formal model drives the creation of the UI, rather than UI designs being used as the basis for the model as is the case in our work.

Similarly, research which aims to include UI considerations as part of a formal system specification, via the use of Interactors [DH93], [FP90] for example, enables a refinement process for both system and UI together based on the underlying language used to describe the Interactors, which may be Z [BFH95], VDM [DH97] or

[1] With a capital 'C' it is the name of the language whose primary objects are μcharts (lowercase 'c').

any other suitable formalism. This work is driven from the formal side of development and is again different from our proposed methods in that it does not attempt to consider how UIs are developed within a UCD process.

3. 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 firstly examine some principles of refinement individually to consider their suitability as principles for UI refinement. Secondly we consider what part refinement plays in existing UI design processes how this is undertaken and how this relates to the formal principles discussed. We use these as the motivation and basis for developing a formal UI refinement process.

3.1. 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 [Mor98] 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. We therefore need some way of considering usability within our refinement process so we can be satisfied that this does not decrease.

3.2. 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 [Bow05])

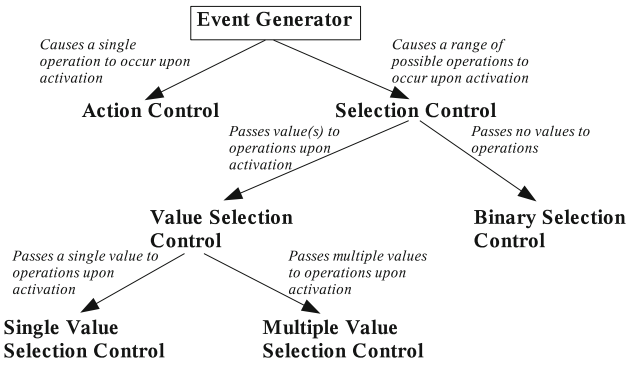

Fig. 1. Event generator hierarchy tree

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 Fig. 1.

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. This is an example of using refinement to guide the transformation process.

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.

3.3. 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 non-essential 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 [Hoa83] 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!

4. Intuitive refinement

Having considered the underlying principles of refinement within a formal process we now describe how refinement is currently included within UI design. UI designers following a UCD process work in an iterative manner. Once they have developed their initial prototype they make incremental changes based on user feedback and development of requirements. As such, we might state that they perform what we will refer to as *intuitive* refinement where they transform their designs using a combination of techniques such as prior experience and design knowledge, design guidelines, house-style, user input, etc. Just as with any refinement the aim is to move from an abstract design towards a concrete implementation, but as with the rest of the UCD UI design process this intuitive refinement is informal.

For example, consider the two UIs given in Fig. 2 which are for a simple application designed to display three different shapes. The left hand side shows a paper prototype and the right hand side an implementation. The visual

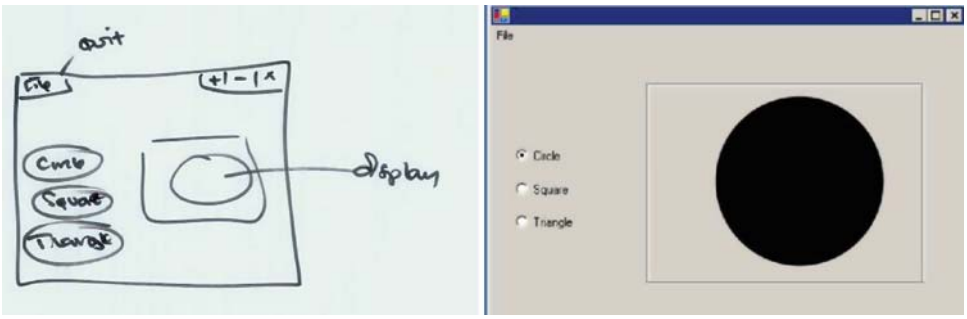

Fig. 2. Prototype and implementation of shape application

appearances of the two UIs are close enough that the designer may have an intuition that the implementation correctly refines the design. The paper prototype shown is virtually a completed design, and as such creating the implementation has required little by way of design changes (the most significant being the choice of radio buttons as the controls) and is mostly the result of implementing the design in code. In this example the intuition is supported by the simplicity of this application. It is possible that this may be equally true of a more complex application (particularly if the design approach is incremental and the gap between the final design and the implementation itself is small) but this is not always the case.

While some aspects of refinement can be supported by intuition, such an approach does not provide any guarantees of correctness, and as we will show shortly, may also lead to errors. Once we have determined exactly what our requirements for UI refinement are we will show how, if at all, intuitive refinement supports these requirements, and perhaps more importantly how a more formal approach is beneficial to UI designers as well as formal practitioners.

5. 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.) [BR06b], [BR06a]. 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.

5.1. Formal models of UIs

In order to identify the properties of UIs that we are interested in for refinement purposes 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 [BR06a] and [BR06b], 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.

6. Informally describing refinement

6.1. 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 [BR06a]. One of these types of equivalence is functional equivalence which has the following definition:

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 at this stage we are still 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.

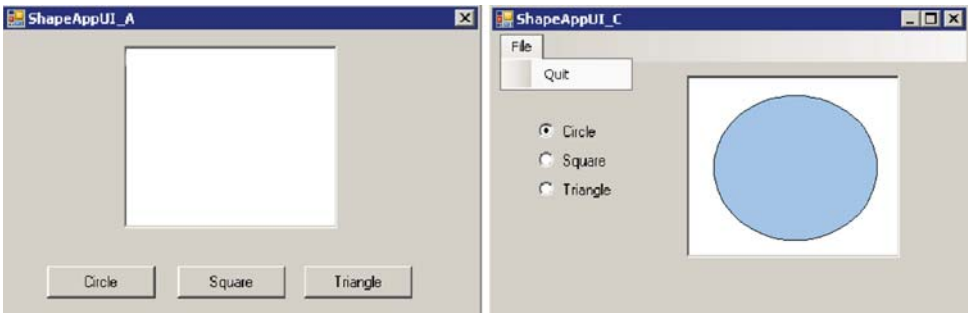

Fig. 3. Design UI_A and UI_C

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.

As an example of these considerations we present two UI designs and their presentation models. The UIs are for the application we described in Sect. 4 which allows a user to display different shapes. UI_A, on the left of Fig. 3 is the original design and UI_C, on the right of Fig. 3 is a suggested refinement. The presentation models for the designs are:

UI_A is
(CircleButt, ActionControl, (S_ShowCircle)),
(SquareButt, ActionControl, (S_ShowSquare)),
(TriangleButt, ActionControl, (S_ShowTriangle)),
(ShapeFrame, SValueResponder, (S_DisplayShape)),
(QuitButt, ActionControl, (I_QuitApp))

UI_C is
(CircleRB, RadioButton, (S_ShowCircle)),
(SquareRB, RadioButton, (S_ShowSquare)),
(TriangleRB, RadioButton, (S_ShowTriangle)),
(ShapeFrame, SValueResponder, (S_DisplayShape)),
(FileMenu, Container, ()),
(QuitMenuItem, ActionControl, (I_QuitApp)),
(QuitBox, ActionControl, (I_QuitApp)),
(MinBox, ActionControl, (I_MinWindow)),
(MaxBox, ActionControl, (I_MaxWindow))

From the presentation models we can derive the following:

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

Comparing these sets shows us that:

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

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.

6.2. 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 Fig. 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 not selected a widget category which is not a child of the previous one, i.e. become incorrectly less abstract.

In the example given in Fig. 3, UI_A has standard buttons whereas UI_C 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 [Gno02] 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.

6.3. 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 earlier design.

6.4. 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 [BR07a]). 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. So the new PIM must be at least as reachable as the previous PIM, although if the original was not totally reachable then it is acceptable for the new PIM to be *more* reachable.

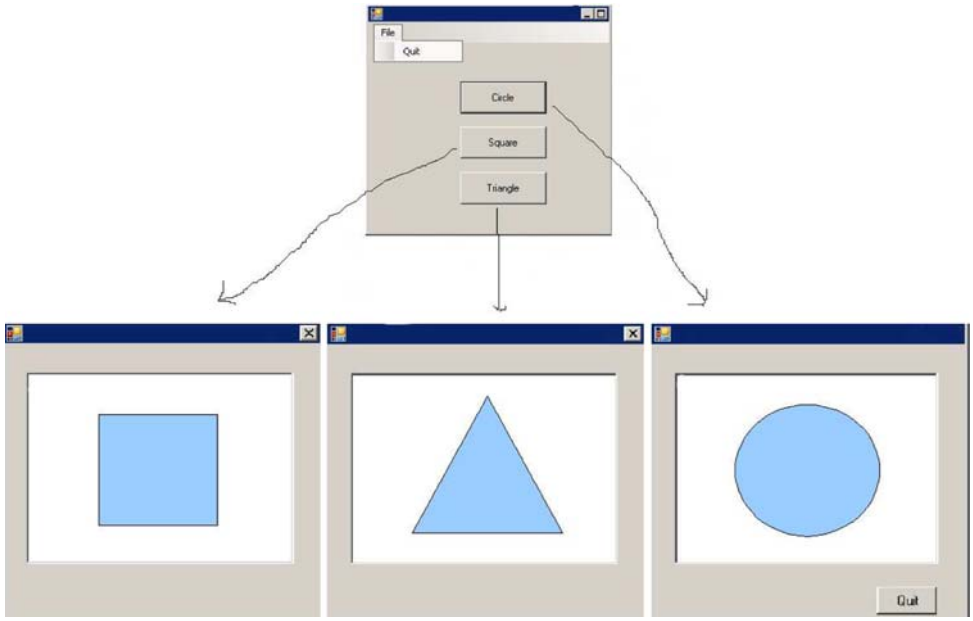

Fig. 4. Design UI_{Multi}

To demonstrate this consider another possible UI for the shape application which we give in Fig. 4. The presentation model for UI_{Multi} is:

```
UIMulti is MainWin : SquareWin : CircleWin : TriangleWin
MainWin is      (SquareCtrl, ActionControl (I_OpenSqrWin, S_ShowSquare))
                (TriangleCtrl, ActionControl (I_OpenTriWin, S_ShowTriangle))
                (CircleCtrl, ActionControl, (I_OpenCrcWin, S_ShowCircle))
                (MinCtrl, ActionControl, (I_MinWindow))
                (MaxCtrl, ActionControl, (I_MaxWindow))
                (FileMenu, Container, ())
                (QuitMI, ActionControl, (I_QuitApp))
SquareWin is    (ShapeFrame, SValueResponder, (S_ShowSquare))
                (CloseBox, ActionControl, (I_OpenMainWin))
TriangleWin is  (ShapeFrame, SValueResponder, (S_ShowTriangle))
                (CloseBox, ActionControl, (I_OpenMainWin))
CircleWin is    (ShapeFrame, SValueResponder, (S_ShowCircle))
                (QuitButt, ActionControl, (I_QuitApp))
```

The PIMs for both the original design, UI_A from Fig. 3, and UI_{Multi}, are given in Fig. 5. The PIM for UI_A is straightforward as there is only one PModel in the presentation model giving us a PIM with a single state (which is both the start state and a final state). The PIM for UI_{Multi} has four PModels, which produces a PIM with four states, where MainWin is the start state, and both MainWin and CircleWin 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_{Multi} 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 any other state from the *CircleWin* state. 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_{Multi} does not maintain the usability of UI_A and is, therefore, not a suitable refinement.

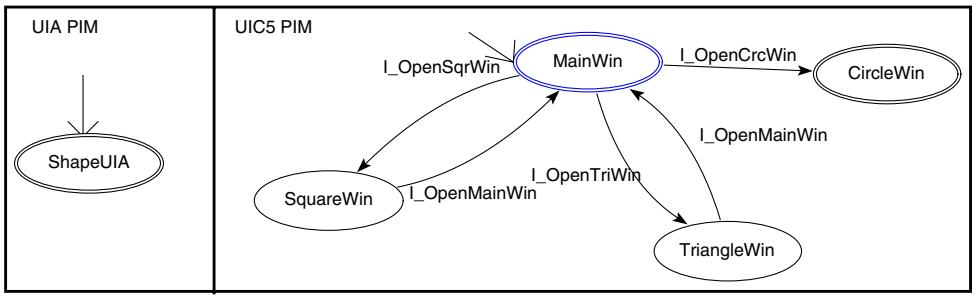

Fig. 5. PIMs for UI_A and UI_{Multi}

7. 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.

In Sect. 4 we described the process of intuitive refinement whereby UI designers rely on expertise and guidance from design rules, etc. to implement their UI designs. We can now compare this with the use of contractual utility and start to show how this enables potential errors to be avoided.

If we consider again our informal requirements for UI refinement we might have some understanding about which of these are more likely to be supported by intuitive refinement than others. Refinement of widgets requires that we correctly instantiate abstract widgets with actual widgets, where correctly means that we follow the widget category hierarchy tree. An experienced designer is likely to have a good understanding of widget usage such that they may be highly likely to have a correct intuition about this step. Less experienced designers, however, may not be so familiar with the importance of correct choice of widgets. Reduction of abstraction (for example by transforming the paper prototype into a coded implementation) may be supported intuitively, especially in cases such as our previous example where the two UIs are almost identical visually, but as we shall see shortly this is not always the case. Defining layout and appearance is, by its very nature, an intuitive task. As we have already seen this is a requirement that is not supported by the models as it relates to the aesthetics of the design. We might state that this requirement, therefore, is always satisfied by intuition. For each of these three requirements we can see that when such changes happen towards the end of the design process, i.e. when we are fine-tuning the design, they are most likely to be supported intuitively as we are making very small, incremental changes.

Our remaining refinement requirements, however, are least likely to be supported intuitively. Maintaining usability and retaining contractual utility are qualities which are hard to observe using the visual appearance of UI designs and as such these are exactly the sorts of properties best suited to examination by way of the formal models. In particular, these are the most likely properties to lead to incorrect intuitive refinement, that is, where designers believe a refinement has occurred but one, or both, of these requirements has been broken.

One of the problems with intuitive refinement is that as applications become more complex, and particularly as numbers of different windows, dialogues, etc. begins to increase, then it becomes more likely that our intuition

may not be correct. Recall, for example, the shape application UI with multiple windows which we gave in Fig. 4. Whilst this may appear to intuitively refine the original $ShapeUI_A$ design (shown in Fig. 3), there is in fact a usability weakening (caused by the lack of a close/return button in the circle window) which we could discover by the reduction of reachability in the relevant PIM, but which otherwise is easy to miss. This is an example of what is still a small and simple application where we might be deceived by intuitive refinement.

In addition, there are cases where it is unlikely that any notion of intuitive refinement between designs will exist at all. In Sect. 13 of this paper we will discuss how different modes of interaction lead to UIs with very little in common visually which makes such intuition difficult.

In summary, although there are parts of the refinement process which may be intuitive to designers it is unlikely that this will extend to complex, real-world problems. The informal refinement description we have given enables us to start to provide some guarantees that such intuition lacks, as well as support less experienced designers who may not have the same level of intuition.

Our next step then is to 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. Before we can do this, however, we first must decide how to combine our UI and system descriptions in order to be able to consider the complete application (which we describe as the composition of the system and UI and refer to as $UI \parallel Sys$) for refinement purposes).

8. Composing the UI and system

Our presentation models and PIMs enable us to formalise design artefacts and start to integrate these with a formal system specification (we provide full details on this in [BR07b]). However, for refinement purposes we would like to be able to describe $UI \parallel Sys$ using a common formalism. In earlier work on presentation models, [BR06a] and [BR06b], we have discussed how they can be integrated using the Z language [Iso02]. However, converting our UI models into Z and integrating them into a single formal specification with the underlying system is neither a straightforward nor satisfactory approach. Although presentation models and PIMs are formal, they retain a simplicity (and in the case of PIMs a visual appearance which is easily mapped to the UI design) which enables them to be easily understood by UI designers without knowledge of a specialist formal language, such as Z.

Our solution is to 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.

μCharts is a visual, Statechart-like language used for describing reactive systems, which has both a logic and a refinement theory [GRR02], [Ree05]. 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 (in fact, we already use the visual conventions of μcharts for representing PIMs for this very reason). The 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. However, as we already have a representation of the UI as a μchart, namely the PIM, we can use this in composition with a μchart of the underlying system. It can be shown via simple examples of total UI models in μCharts that this abstraction (the PIM) is in fact the same thing as modelling the entire UI. 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.

Returning to our earlier example of the shape application, we now show how we would model this along with related parts of the underlying system as a composed μchart. In Fig. 3 we presented a UI design for a simple shape application, UI_A. We now give the composed μchart for the PIM of this UI, along with the underlying system in Fig. 6. Note that in general we do not need to include the entire system in a single μchart, but rather we model the parts of the system related to the UI functionality and compose this with the PIM μchart. The modularity of

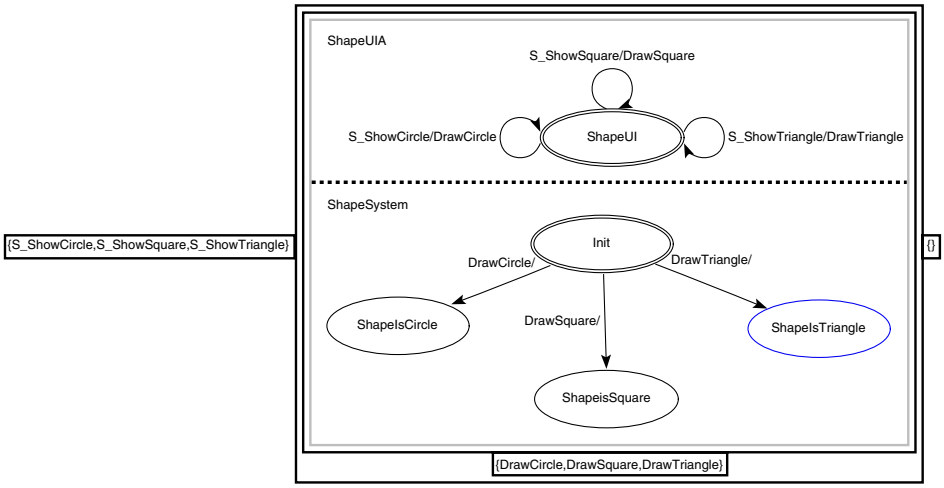

Fig. 6. Composed chart for shape application

μCharts means we can model related parts of the system and UI as composed charts and subsequently further compose these with charts representing other parts of the system.

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, $\{S\_ShowCircle,\ S\_ShowSquare,\ S\_ShowTriangle\}$, 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 which for our considerations we can imagine 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, $\{DrawCircle,\ DrawSquare,\ DrawTriangle\}$. We have also extended the original PIM to include $S\_Behaviours$ which appear as transitions within the UI μchart (rather than just the $I\_Behaviours$ as previously), this enables us to include the communication between UI and system (which is, of course, via these $S\_Behaviours$) and to include the visual representation of the relation between $S\_Behaviours$ and system operations in the chart. We don't rely on the underlying relation between states in the PIM and component presentation models (and their behaviours) to understand the interactivity as we have previously with the PIM.

The behaviour of the described system then is that it starts in the states represented 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 $S\_ShowCircle$ 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 $S\_ShowSquare$ signal is input when the charts are in their initial states then the transitions to $ShapeUI$ and $ShapeIsSquare$ are made, and likewise for input of $S\_ShowTriangle$. 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 [Ree05] 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. Before we move on to examining trace refinement we first present a description of what we will refer to as *validity* which provides constraints on $UI \parallel Sys$ and the composed μcharts we use to describe them.

9. Validity

In our previous section we have discussed how we can bring together UI and system descriptions into a single model. However, we also want to be sure that when we do this the two parts of the composition will interact

correctly. We will now be more specific about our expectations of the UI and system specifications and how they are related, and define what we will call *valid* applications. In order for our specifications to define a valid application they must meet certain criteria. We will explain what these criteria are and then show how these affect the way in which we model system and UI compositions using μCharts.

In general, we say that an application is valid when the UI specification (by which we mean anything from a design to an implementation) and the system specification (any formal specification at any level of abstraction up to and including an implementation) are consistent with each other.

Recall that within our UI designs we identify two types of behaviours, $I\_Behaviours$ and $S\_Behaviours$. In our earlier research (see [BR06b] for example) we have explained how we can create a relation between presentation models and system specifications which shows which behaviours of the UI are intended to make operations of the system available to a user. For example, in our shape example the UI has three $S\_Behaviours$, ($S\_ShowCircle$, $S\_ShowSquare$ and $S\_ShowTriangle$) which leads to the following relation (which we call the presentation model relation, or PMR):

$S\_ShowCircle \mapsto DrawCircle$
$S\_ShowSquare \mapsto DrawSquare$
$S\_ShowTriangle \mapsto DrawTriangle$

We define a valid application as one where the PMR is a total, onto, many-to-one relation (the domain of the target set being only system operations which should be available to users rather than *all* operations). The expectation is, therefore, that we only compose UI and system descriptions designed for the same application (there is little point in composing arbitrary UIs and systems) and we can show that there is consistency in their interactive behaviour, i.e. the UI provides a way for a user to interact with the system operations and both the UI and system agree on what these available operations are.

We then need to ensure that our $UI \parallel Sys$ μcharts correctly capture our notion of validity. This relies on correct communication between user and UI, and UI and system. We can give a set of requirements which must hold true in order for a μchart to be considered valid (i.e. it is a correct model of a valid application).

$UI \parallel Sys$ comprises two sequential charts, one which models the UI and the other which models the underlying system. The chart representing the UI is an extended PIM, and as such has all of the usual constraints which ensure that a PIM is correct (for example by ensuring that all input signals on transitions are behaviours which can be found in the presentation model related to the state the transition is leaving).

First we describe how a valid $UI \parallel Sys$ is correctly modelled using μCharts to ensure that the overall behaviour captured by the validity conditions are also true of the μchart. The behaviours described in presentation models consist of $S\_Behaviours$ and $I\_Behaviours$. The presentation model relation (PMR) consists of related pairs of behaviours and operations (from the system specification) which describe the UI and system interaction. These are represented within μcharts as follows:

- Each $S\_Behaviour/Operation$ pair appears as the guard/action pair of at least one transition in the sequential UI μchart
- Each $I\_Behaviour$ appears as the guard on at least one transition in the sequential UI μchart
- All $Operations$ related to $S\_Behaviours$ appear in the feedback set of the composed $UI \parallel Sys$ μchart

In addition, we must ensure that the user's ability to interact with the application (via the UI) is correctly represented. We use the input and output interfaces of μcharts to control this. The requirements are that:

- Both sequential μcharts of the composition have their natural interfaces (i.e. there is no filtering or restricting of signals)
- The input interface to the composed μchart is exactly the natural interface of the sequential UI μchart

If we return to the μchart for the shape application which was given in Fig. 6 we can now check that this is a correct model of a valid application.

To begin with we examine the PMR to ensure that the relation defines a valid application (if we do not start with an application which is valid we cannot give any guarantees about our model). We then check each of the requirements given above to make sure the model is correct. We begin by considering the $S\_Behaviour/Operation$ pairs in the PMR, these are:

$S\_ShowCircle \mapsto DrawCircle$, $S\_ShowSquare \mapsto DrawSquare$ and $S\_ShowTriangle \mapsto DrawTriangle$

If we examine the sequential chart called $ShapeUIA$ we see that each of these appears as a guard/action on a transition and our first requirement is therefore satisfied. Next we consider each of the $I\_Behaviours$ in the

presentation model. In fact in this example we do not have any $I\_Behaviours$ as the PIM has a single state only (based on the UI which has a single window only) and so we have nothing further to check for this condition. We next check that all of the $Operations$ related to $S\_Behaviours$ in the PMR appear in the feedback set between the composed charts. The feedback set in this case is $\{DrawCircle, DrawSquare, DrawTriangle\}$ meeting the requirement.

Having satisfied ourselves that we have built the μchart in accordance with the requirements which ensure it correctly models $UI \parallel Sys$ we now turn our attention to the behaviour of the chart with respect to the inputs and outputs from the environment (or user). The two sequential charts which make up the composition are shown without any explicit input and output interfaces. This means that we assume them to have their natural interfaces (there is no restriction on any input or output signals) such that the natural input interface to $ShapeUIA$ is $\{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$ and the natural output interface is $\{DrawCircle, DrawSquare, DrawTriangle\}$, while the natural input and output interfaces of $ShapeSystem$ are $\{DrawCircle, DrawSquare, DrawTriangle\}$ and $\{\}$ respectively. The input interface to the composed chart is $\{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$ i.e. it is the same as the natural interface to the sequential UI chart. Our composed chart, therefore, meets all of the requirements we have described and we are satisfied it correctly describes a valid application.

We have then shown how we can define what is a valid $UI \parallel Sys$ based on methods of interaction between systems and UIs. Subsequently we have described how we model these compositions using μcharts which follow a prescribed pattern to ensure they correctly capture the behaviour we require. Finally we have shown how we also enforce the required interaction possibilities between user and UI and UI and system by way of μcharts interfaces. We now move on to describe the trace refinement theory for μcharts and use this, in conjunction with our valid, composed μcharts, to examine the UI refinement we have described informally.

10. Trace refinement

The refinement theory of μCharts can be described in terms of traces. 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 which 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 Fig. 6. One possible trace for this chart is:

$$(\langle\{S\_ShowCircle\}\rangle, \langle\{\}\rangle)$$

another possibility is:

$$(\langle\{SS\_ShowSquare\}\rangle, \langle\{\}\rangle)$$

and a third is:

$$(\langle\{SS\_ShowTriangle\}\rangle, \langle\{\}\rangle)$$

Note that the output trace set is empty even though the transitions which occur on input of any of the signals described do have output signals. This is because of the output interface to the composed chart, which is empty. That is, no signals will be emitted to the external environment. For the purposes of this description we will rely on the $do - nothing$ semantics for μCharts (although for refinement purposes we in fact rely on the total-chaos interpretation). What this means is if we are in some state and a set of signals is seen for which there is no defined behaviour, then the chart will do nothing. We can therefore generalise the traces described above under the *do-nothing* interpretation, and say that there are exactly three possible traces for this chart which are:

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

where ϵ represents any sequence of input sets with elements drawn from $SShowCircle$ $SShowSquare$ and $SShowTriangle$ 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

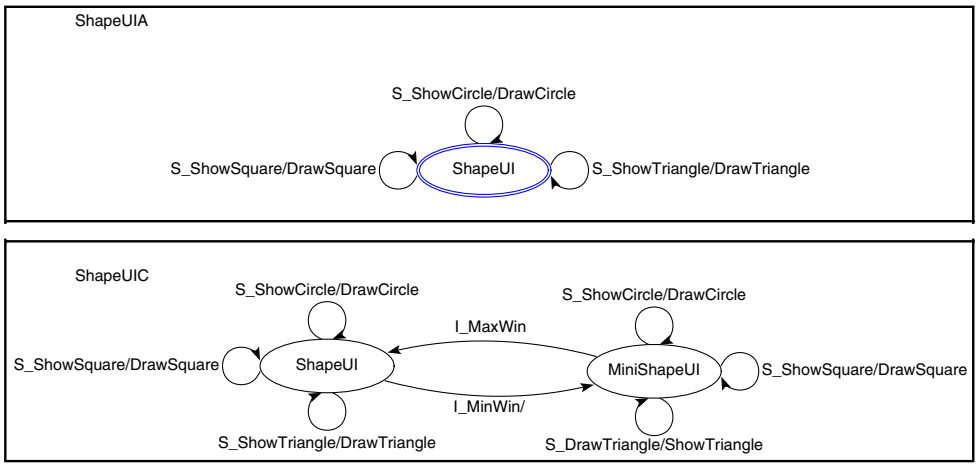

Fig. 7. Sequential charts for UIA and UIC1

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 assume such nondeterminism does not exist. In addition, we will rely on the $total - chaos$ semantics of μCharts for refinement, this is where an input signal which has no defined behaviour leads to the chart behaving chaotically, which means any traces are possible (leading to nondeterminism). The second type of refinement 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 in our informal definition of UI refinement 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

$$C \approx_I A =_{def} \forall i;\ o \bullet (i_{\triangleright(in_C)}, o) \in [[C]] \iff (i_{\triangleright(in_A)}, o) \in [[A]] \\ \wedge\ out_C = out_A$$

$$C \approx_O A =_{def} \forall i;\ o \bullet (i, o_{\triangleright(out_C)}) \in [[C]] \iff (i, o_{\triangleright(out_A)}) \in [[A]] \\ \wedge\ in_C = in_A$$

where i is the sequence of input signals and o the sequence of output signals and $i_{\triangleright in_x}$ restricts the range of the sequence i to the signals in the set in_x (and similarly for $o_{\triangleright out_x}$). 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.

10.1. Example

Returning again to our shape application and the possible UI designs for that system we will give an example of using trace refinement. In Fig. 3 we gave two possible UI designs for the shape application, UI_A and UI_C. In Fig. 7 we give the sequential μcharts for each 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 are:

$$in_A = out_A = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle, DrawCircle, DrawSquare\}$$
$$in_{C1} = out_{C1} = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle, DrawCircle, DrawSquare, \\ I\_MinWin, I\_MaxWin\}$$

In our earlier discussions on contractual utility we had stated that UI_C was an acceptable replacement for UI_A 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 themselves that this is true.)

11. Monotonicity of refinement

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 Fig. 6 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. We have just shown how we can refine the UI in isolation (from UI_A to UI_{C1}) and we would like to be sure that if we compose the new μchart for the UI with the same system chart then we similarly have a satisfactory refinement of the whole application.

In his description of refinement for μCharts [Ree05], Reeve shows that the composition of μcharts is monotonic with respect to refinement, but requires additional side-conditions to hold. The three side-conditions are:

$(SC1)$ *is* $A_\Psi \sqsupseteq^T_{tf} C_\Psi$
$(SC2)$ *is* $out_A \cap \Psi_{AB} = out_C \cap \Psi_{CB}$
$(SC3)$ *is* $out_A \cap out_B = out_C \cap out_B$

where $(A \parallel B)$ is the original composed chart and $(C \parallel B)$ is the new composed chart, and A has been refined to C but B remains unchanged. The first side-condition imposes a stricter refinement condition on charts A and C by referring to them in the context of the feedback set Ψ (i.e. the signals in the feedback set are considered as the fixed inputs and outputs permissible for each chart) and stating that in this context a refinement must still hold. This has two effects, firstly it ensures that all state/input pairs with defined behaviour are retained (so the chart cannot become less reactive) and secondly it ensures that output behaviour with respect to feedback does not change (that is, output signals which are also in the feedback set must be preserved). The second side-condition ensures that the set of output signals which are also in the feedback set does not change during refinement (by interface restriction for example), so this intersection cannot get smaller or larger and preserves communication between the composed charts. Finally, the third side-condition ensures the intersection of output signals from each composed chart is the same, which ensures that output refinement of one of the sequential charts does not enable the new chart to control the environment with respect to any of the signals in the chart it is composed with.

By ensuring that our refinement for $UI \parallel Sys$ meets these side-conditions we can also ensure that the refinement will be monotonic. However, the side-conditions have a more interesting part to play in relation to $UI \parallel Sys$ and our validity requirements. Not only do we want refinement for $UI \parallel Sys$ to be monotonic, but we also want to be certain that if we begin with a valid $UI \parallel Sys$ then refinement will also preserve validity. We show next how the correspondence between the validity conditions and the monotonicity side-conditions enables us to be satisfied that this is indeed the case.

12. Monotonicity and validity

In order to preserve validity during refinement, each of the requirements we have given in Sect. 9 must continue to hold after the refinement. In fact, we show now how each of these requirements is guaranteed to be preserved, either by the refinement rules, or the monotonicity side-conditions. That is, if we monotonically refine a valid $UI \parallel Sys$ then the resulting chart is guaranteed to be valid also.

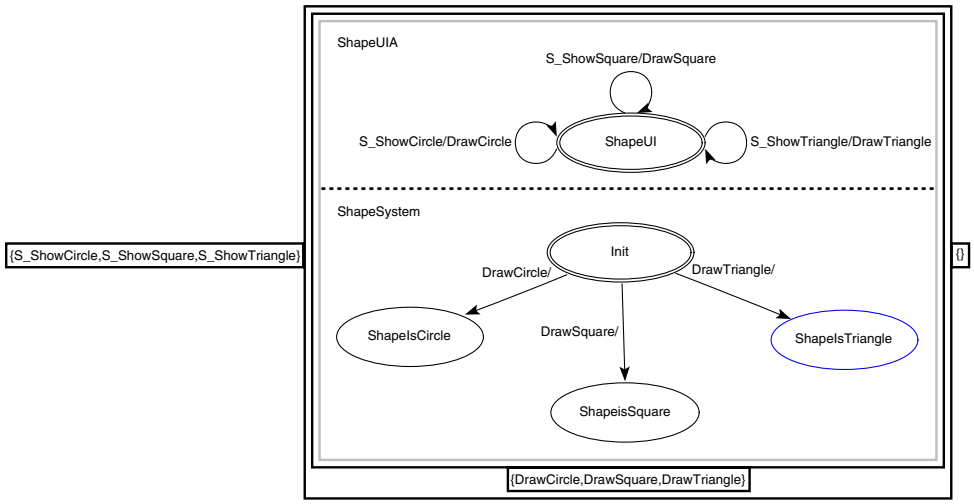

Fig. 8. Composed chart for shape application

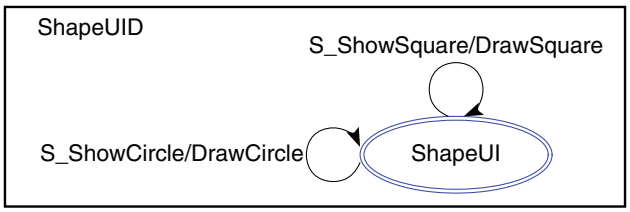

Fig. 9. ShapeUID

We will examine, in turn, each of the criteria for ensuring μCharts are correct models of valid applications and show how these criteria are maintained either by refinement requirements or the monotonicity side conditions.

- Each $S\_Behaviour/Operation$ pair appears as the guard/action pair of at least one transition in the sequential UI μchart.

First we consider all of the changes we could make to a sequential μchart to break this validity condition. For each $S\_Behaviour/Operation$ pair there are exactly three ways to break this condition. We can remove all of the transitions which satisfy the requirement. We can remove the $S\_Behaviour$ from the guard of all satisfying transitions, or we can remove the $I\_Behaviour$ from the actions of all satisfying transitions. Finally we could change the input or output interface to filter the signals on these required transitions. We can examine each of these in turn using our Shape example. For ease of reference we repeat the composed μchart for the shape UI and system in Fig. 8. Suppose we try to refine $ShapeUIA$ by removing one of the loop transitions (which would then break validity). The chart $ShapeUID$ given in Fig. 9 is an example of this where the transition which occurs on input of $S\_ShowTriangle$ has been removed. However, $ShapeUID$ is not a refinement of $ShapeUIA$ as it will now behave chaotically on input of $S\_ShowTriangle$, this leads to traces of $ShapeUID$ which are not traces of the original chart. It is always the case that removing such transitions from states in this way means we do not get a refinement as we will get chaotic behaviour from the (now) undefined inputs. We can prevent this by restricting the input interface, but we will show shortly that this leads to other problems. Similarly, if we remove the $S\_Behaviour$ as an input signal to one of the transitions (for example $S\_ShowTriangle$ from the loop transition $S\_ShowTriangle/DrawTriangle$) we get new chaotic traces for the chart, which are not in the original and therefore refinement does not hold. Removing the output of a transition will likewise break the refinement, but for different reasons. If we remove $DrawTriangle$ from the transition we do not get chaotic behaviour, but introduce the trace:

$$(\langle\{S\_ShowTriangle\}\rangle,\ \langle\{\}\rangle)$$

which is not a trace of the original chart.

Changing the output interface to the sequential *ShapeUIA* chart to remove, for example, the signal *DrawCircle* also introduces the trace

$$(\langle\{S\_ShowTriangle\}\rangle,\ \langle\{\}\rangle)$$

and is the same as above and therefore not a refinement. Removing $S\_ShowTriangle$ from the input interface will once again lead to chaotic behaviour as we cannot say what will happen when no input is seen.

It is not always the case that removing or changing transitions in the way we have just shown will cause refinement to fail for general μcharts. It is partly the nature of the conditions we have placed upon our charts to begin with that leads to this effect. μcharts which describe valid applications are a restricted subset of all possible μcharts, and as such refinement possibilities are similarly restricted.

We can consider this more generally in terms of two arbitrary sequential charts for the same UI, which we will refer to as A and C, where A is known to part of a valid composition. Recall that the PMR for a valid application is a many-to-one relation. Taking this in conjunction with the first validity requirement means that we know any transition in chart A with a guard $SBeh$ will *always* have an action IOp. In addition the signal $SBeh$ will be part of the natural input interface to A. Now we consider the definition given in [Ree05] of behavioural refinement for μcharts which is:

For arbitrary charts A and C we have:

$$C \sqsupseteq_b A =_{def} \forall i;\ o \bullet in_C = in_A \wedge out_C = out_A \wedge (i_{\triangleright(in_C)}, o_{\triangleright(out_C^{\perp})}) \in [[C]] \Rightarrow (i_{\triangleright(in_A)}, o_{\triangleright(out_A^{\perp})}) \in [[A]]$$

In order for C to break validity there must be no possible transition with $SBeh/Op$ as the guard and action. Given that A and C have the same input interfaces (given in definition above) and A has a natural input interface which therefore must include $SBeh$, it is always possible for $SBeh$ to appear as an input signal to C. There are then three possibilities for traces resulting from such an input:
$(\langle\{SBeh\}, \{\}\rangle)$ is a trace of C, but this breaks the definition of behavioural refinement as there is no such trace in A; or $(\langle\{SBeh\}, \{\alpha\}\rangle)$ is a trace of C (where α is any signal other than Op), but again this breaks the definition of behavioural refinement as there can be (due to PMR) no such trace in A; or there is no defined behaviour for input signal $SBeh$ in C in which case any number of (chaotic) traces are possible, which again will not be traces of A.

We see, therefore, that for any general case of a sequential chart which is part of a valid composition, the definition of behavioural refinement, interface refinement, and the monotonicity side-conditions guarantees that a monotonic refinement of the μchart will result in a composition which meets the first validity requirement.

- Each $I\_Behaviour$ appears as the guard on at least one transition in the sequential UI μchart.

Again we consider this in the general case of two arbitrary charts A and C.

In order to break this requirement there must be no transition in C which has $IBeh$ as its guard. As we saw in the previous example, the natural interface condition means that $IBeh$ is in the input interface of A and therefore, due to the refinement definition, is also in the input interface of C. If there is no defined behaviour for C on the input of signal $IBeh$ (which is the case if this requirement is broken) then this results in any number of (chaotic) traces for C which are not traces of A and hence there is no refinement. We see, therefore, that the definition of behavioural refinement guarantees that a monotonic refinement of the μchart will result in a composition which meets the requirement.

In the case of the transitions with $IBeh$ as a guard there are no corresponding output actions, so it may appear that with a combination of both behavioural and interface refinement we could remove such transitions (by removing the transition and filtering out the input signal). Consider the chart given in Fig. 10. Suppose we remove the transition from Min to Max and also restrict the input interface by removing the signal I_Max. Now if the signal I_Max is input when the chart is in state Min it will not be seen (it is filtered out by the interface) but the chart will not behave chaotically as there is a defined transition for the signal not being seen in this state (the loop transition with $-I\_Max$ as the guard). It might appear therefore that this would be a correct refinement which would also break our validity condition. However, the definition for both behaviour and interface refinement prevents the removal of input (or output) signals and behavioural changes in one step, rather there must be intermediary charts which satisfy one of four cases. For the example we have given the applicable case is:

$$in_C \subset in_A \wedge out_A \subseteq out_C$$

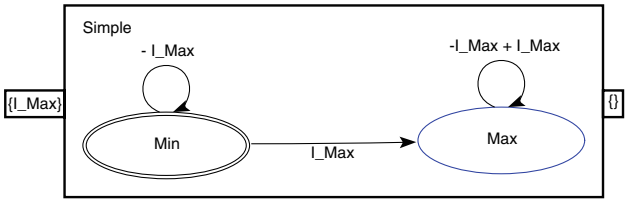

Fig. 10. Simple chart

Now in order for a behavioural and interface refinement to hold we must satisfy the following condition:

$$\exists B, B' \bullet C \approx_I B' \wedge B' \sqsupseteq_b B \wedge B \approx_O A$$

However, no such chart as B′ exists. This is because a chart that is behaviourally the same as C but which has the input signal I_Max in its interface (which is what is required of a chart B′) reintroduces the chaos seen when input signal I_Max is input when the chart is in the Min state and is therefore not a refinement and so the overall refinement does not hold. The rules for both interface and behavioural refinement are intended to prevent exactly the action we have performed (namely removing both signals and transitions in one step), and therefore the validity requirement is preserved.

- All $Operations$ related to $S\_Behaviours$ appear in the feedback set of the composed UI || Sys μchart.

The only way to break this requirement is to remove one, or more, of the related $I\_Operations$ from the feedback set. However, the second monotonicity side-condition states:

$$\Psi_{AB} \cap Out_A = \Psi_{CB} \cap Out_C$$

We also know, from the refinement definition, that:

$$Out_A = Out_C$$

If both output interfaces are the same and the intersection with feedback is also the same then it follows that feedback for both charts must also be the same. This prevents us from removing any of the signals from feedback and therefore the monotonicity side-condition guarantees that this requirement is met.

- Both sequential μcharts of the composition have their natural interfaces (i.e. there is no filtering or restricting of signals).

Behavioural refinement requires that input and output interfaces remain unchanged, therefore we only need consider interface refinement for this condition. For general μcharts we can restrict (or expand) chart interfaces and maintain a refinement. However, the types of charts we create for a valid application means that changing the interfaces is not possible as a means of refinement. This is partly due to the fact that validity requires that we start with the natural interface of the chart and therefore restricting the interface will always directly affect transitions and cause chaotic behaviour (breaking refinement) or reducing the responsiveness of chart behaviour (breaking SC1). Expanding the interface leads to chaotic behaviour as there can be no defined transitions for the new signals in a chart with already has its natural interface (rather than in general chart theory where we may start with a chart with a restricted interface and refine it to another chart with its natural interface).

Restricting the output interface is not permitted due to SC2 and expanding the output interface can have no effect (again due to our starting point being the natural interface where all output signals are already present).

- The input interface to the composed μchart is exactly the natural interface of the sequential UI μchart.

As above, the inability to restrict (or meaningfully expand) the interfaces ensures this is maintained.

We have then shown that any monotonic refinement of a valid composed UI || Sys μchart will result in a μchart that is similarly valid (as the refinement preserves the validity requirements). In order to understand why this is the case we can consider the relationship between the monotonicity side-conditions and the validity requirements. The *purpose* of the validity requirements is to ensure that the intended behaviour of the application is described correctly in the μchart. By this we mean that the communication between user and UI, and UI and system, are described in the same way as in the PIM and PMR. The first two requirements ensure that the necessary transitions are present in each of the sequential charts to allow correct communications (that is,

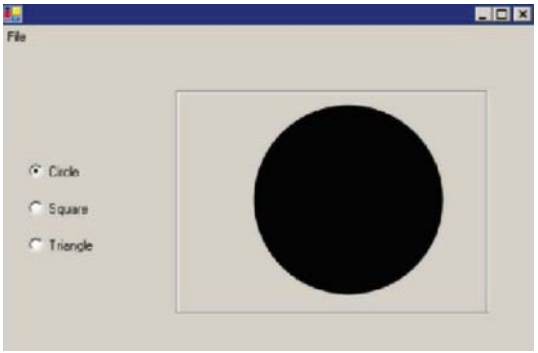

Fig. 11. Shape application

transitions which accept or output the signals representing behaviours and operations of the PMR). The final three requirements describe conditions on the feedback set, Ψ, and the input and output interfaces which ensure that the necessary communication is possible. The requirements therefore ensure the correct transitions are present for communication between UI and System and that it is correctly described in $UI \parallel Sys$. The refinement theory of μCharts ensures that the correctness of the transitions is preserved and the monotonicity side-conditions ensure that the communication is preserved. In fact, the exact purpose of the side-conditions is maintaining the communication between composed charts by restricting the allowable changes to feedback and interfaces which may affect this.

13. Horizontal and vertical refinement

In our discussions about refinement so far we have assumed that we are refining some abstract description of a UI (or total system) in order to get closer to a final implementation of that UI (or system). We will refer to this as horizontal refinement. At each iteration we move a step closer to the final implementation. We have defined (both informally and formally) this type of refinement and shown, using μCharts, that it is both monotonic and valid (with respect to composition).

However, we can also consider a different type of refinement, which we refer to as vertical refinement. In this case we are not considering designs for the same system at different levels of abstraction, but rather designs for different versions of the same system. Such is the case when we develop one system which is required to run on different platforms as well as in cases where we are upgrading, or extending an existing system.

In this case while there is still a correspondence between the user requirements for both of the systems, the nature of the different platforms, or the extensions required, mean that we cannot rely on the relationship between respective I and S behaviours from the presentation models as we have for horizontal refinement. While we want to be sure that all of the original requirements are met, the design of the system may be very different (due to hardware constraints, etc.) meaning there is no direct correspondence between either the system specifications or the presentation models of the UI designs. This also means that the other refinement requirements we could consider using the models (namely preservation of usability and correct widget choice) are also not applicable.

As an example we return again to the shape application. Our original version had a UI design prototype which we now repeat in Fig. 11. Now consider the design given in Fig. 12. This is a UI for the same application, but this time designed to run on a PDA. The user requirements for both systems are the same, namely to allow the user to display either a circle, square or triangle on the screen. However, due to the differing nature of the hardware and nature of interaction, there is a difference in the design of both the system and UIs for the two versions. The presentation model for the UI design of Fig. 11 is:

ShapeApplication is
(CircleCtrl, ActionControl, (S_ShowCircle)),
(SquareCtrl, ActionControl, (S_ShowSquare)),
(TriangleCtrl, ActionControl, (S_ShowTriangle)),
(ShapeFrame, SValueResponder, (S_ShowCircle, S_ShowSquare, S_ShowTriangle)),

(FileMenu, Container, ()),
(QuitMenuItem, ActionControl, (Quit)),
(QuitIcon, ActionControl, (Quit)),
(MinWin, ActionControl, (I_MinWindow)),
(MaxWin, ActionControl, (I_MaxWin))

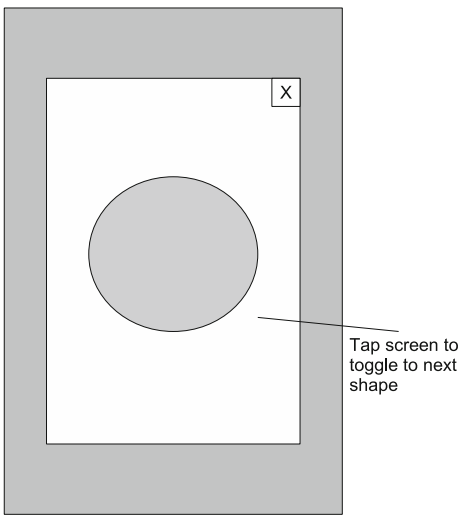

Fig. 12. Shape application for PDA

The presentation model for the design for the PDA version of the application given in Fig. 12 is:

PDAShapeApplication is
(QuitIcon, ActionControl, (Quit)),
(ActiveArea, SValueResponder, (S_ToggleShape))

If we were to try and apply our previously defined conditions on the sets of I and S behaviours to determine UI refinement we would immediately see a problem. We cannot say that PDAShape refines ShapeApp as they do not have the same set of $S\_Behaviours$ and the $I\_Behaviours$ of ShapeApp are not a subset of those of PDAShape. It is not that surprising that there is no relationship between the $I\_Behaviours$ of the two systems. $I\_Behaviours$ are directly related to the interactivity of the system in terms of the hardware and methods of interaction. Where these are different we cannot expect a useful comparison of these behaviours. The differences between the $S\_Behaviours$ is, however, more problematic. Based on their difference we would state that there is no refinement relation between the two UIs. However, given that there is a relationship between the two applications (they are both intended to do the same thing and are developed from the same set of requirements) we might expect that some refinement relation does exist. We introduce vertical refinement which is designed to be used in such situations.

One of the problems we have with this example is that the $ShapeApplication$ model has more behaviours than the $PDAShapeApplication$ model, despite both systems being designed to perform the same tasks. This is due to the nature of interaction with the PDA device which means that instead of having a discrete behaviour to display each different shape there is instead a single behaviour which is repeated a number of times depending on which shape is to be displayed and which shape is currently being displayed.

Refinement between systems where a single operation in the abstract system is replaced by several operations in the concrete system is not unusual in general refinement theory. Derrick and Boiten [DB01] refer to this as non-atomic refinement and describe how a single abstract operation can be replaced with a fixed number of concrete operations. In our example, however, the number of operations in the concrete system is not fixed. For example, upon initialisation both of the systems start up in a state where no shape is displayed, in order to display a

triangle in $ShapeApplication$ the $S\_ShowTriangle$ behaviour is used whereas the $PDAShapeApplication$ requires three uses of the $S\_ToggleShape$ behaviour (assuming the toggle order to be $NoShape \mapsto Circle \mapsto Square \mapsto Triangle \mapsto Circle \ldots$). However, if both systems are in a state where a square is being displayed then in order to display a triangle the $S\_ShowTriangle$ behaviour is again used in $ShapeApplication$ but in $PDAShapeApplication$ a single use of the $S\_ToggleShape$ behaviour is required rather than three as previously. The relationship between the number of operations required for each system is, therefore, state dependent rather than fixed.

Our first step toward a solution is to include this state within the presentation model. When we consider UIs with multiple windows (which are themselves different states of the UI) we describe each state as a component presentation model and then conjoin all of the models (using the ":" operator) to describe the total UI. We take the same approach here and expand the presentation model for the $PDAShapeApplication$ as follows to incorporate the state.

PDAShapeApplication is PDABlank : PDASquare : PDACircle : PDATriangle
PDABlank is
(QuitIcon, ActionControl, (Quit)),
(ActiveArea, SValueResponder, (S_ShowCircle, I_PDACircle))
PDACircle is
(QuitIcon, ActionControl, (Quit)),
(ActiveArea, SValueResponder, (S_ShowSquare, I_PDASquare))
PDASquare is
(QuitIcon, ActionControl, (Quit)),
(ActiveArea, SValueResponder, (S_ShowTriangle, I_PDATriangle))
PDATriangle is
(QuitIcon, ActionControl, (Quit)),
(ActiveArea, SValueResponder, (S_ShowCircle, I_PDACircle))

Now when we compare the sets of $S\_Behaviours$ for $ShapeApplication$ and $PDAShapeApplication$ they are equivalent, reflecting the one-to-one correspondence resulting from the introduction of state.

This suggests how we might begin to develop a weaker version of contractual utility for use in vertical refinement cases than that given for horizontal refinement. This weaker version is based on equivalence of $S\_Behaviours$ only. By expanding presentation models to include state where necessary we can satisfy this weaker version of contractual utility and therefore still consider refinement.

What we do not have, at this stage, is a way to support this weaker version of contractual utility with a formal theory. The trace refinement of μCharts we have described in Sect. 10 is reliant on $I\_Behaviours$, and as they are an integral part of the PIM we cannot simply remove them from our μcharts. We do not attempt to solve this problem here, but consider it as future work for our research.

14. 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 in a formal software development process. That is, the starting point for our UI development is a UCD process, with designs and prototypes being subsequently formalised to enable us to use them within a formal 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 compared this to the sorts of informal and intuitive refinement that traditionally takes place during UI development and shown how our approach both supports and improves such techniques.

We have 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. Using the μChart refinement theory for composed charts we can monotonically refine UI/System pairs and, just as importantly, ensure properties we have described to ensure validity of applications is preserved by the monotonic refinement. Finally we introduced the idea of vertical refinement and gave a weaker definition for our informal contractual utility which can be used to support it.

Throughout this work we have kept 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. The ability to move between visual representations of the UI provided by prototypes, PIMs and μcharts whilst relying on the underlying theory to support the formality does, we believe, support this aim.

References

[BFH95] Bramwell C, Fields RE, Harrison MD (1995) Exploring design options rationally. In: Palanque P, Bastide R (eds) Design, specification and verification of interactive systems '95, Wien. Springer, Berlin, pp 134–148

[Bow05] Bowen J (2005) Formal specification of user interface design guidelines. Masters thesis, Computer Science Department, University of Waikato

[BR06a] Bowen JA, Reeves S (2006) Formal refinement of informal GUI design artefacts. In: Proceedings of the Australian Software Engineering Conference (ASWEC'06). IEEE, New York, pp 221–230

[BR06b] 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

[BR07a] Bowen JA, Reeves S (2007) Using formal models to design user interfaces, a case study. In: HCI 2007: Proceedings of the 21st BCS HCI Group Conference

[BR07b] Bowen J, Reeves S (2007) Formal models for informal gui designs. In: Electronic Notes in Theoretical Computer Science Volume 183, Proceedings of the First International Workshop on Formal Methods for Interactive Systems (FMIS 2006), July 2007, pp 57–72

[BR07c] Bowen J, Reeves S (2007) Refinement for user interface designs. In: Curzon P, Cerone A (eds) The Pre-proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems (FMIS 2007). Lancaster University, UK, September 2007

[DB01] Derrick J, Boiten E (2001) Refinement in Z and Object-Z: Foundations and advanced applications. Formal approaches to computing and information technology, May 2001. Springer, Berlin

[DH93] Duke DJ, Harrison MD (1993) Abstract interaction objects. In: Proceedings of Eurographics '93, Computer Graphics Forum

[DH95] Duke DJ, Harrison MD (1995) Mapping user requirements to implementations. Softw Eng J 1(10):13–20

[DH97] Doherty G, Harrison MD (1997) A representational approach to the specification of presentations. Eurographics Workshop on Design Specification and Verification of Interactive Systems, DSVIS 97, Granada, Spain, June 1997

[FP90] Faconti G, Paternò FM (1990) An approach to the formal specification of the components of an interaction. In: Vandoni C, Duce D (eds) Proceedings of Eurographics '90, North Holland. Springer, Berlin, pp 481–494

[Gno02] Gnome (2002) GNOME Human Interface Guidelines(1.0), 2002. http://developer.gnome.org/projects/gup/hig/1.0/

[GRR02] Goldson G, Reeve G, Reeves S (2002) μ-Chart-based specification and refinement. In: Formal Methods and Software Engineering. 4th International Conference on Formal Engineering Methods, ICFEM 2002, LNCS, vol 2495, Shanghai, China, October 2002. Springer, Berlin, pp 323–334

[HMC00] Hussey A, MacColl I, Carrington D (2000) Assessing usability from formal user-interface designs. Technical Report TR00-15, Software Verification Research Centre, The University of Queensland

[Hoa83] Hoare CAR (1983) Communicating sequential processes. Commun ACM 26(1):100–106

[Iso02] 13568 Iso (2002) Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics. Prentice-Hall International series in computer science. ISO/IEC, 1st edn

[LVM+04] Limbourg Q, Vanderdonckt J, Michotte B, Bouillon L, López-Jaquero V (2004) Usixml: a language supporting multi-path development of user interfaces. In: EHCI/DS-VIS, pp 200–220

[Mic07] Microsoft. Microsoft visual studio, 2007. Microsoft technical pages for the Visual Studio Software

[Mor98] Morgan C (1998) Programming from specifications, 2nd edn. Prentice-Hall, Hertfordshire

[PE02] Puerta A, Eisenstein J (2002) Ximl: a common representation for interaction data. In: IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces, New York, NY, USA, 2002. ACM Press, New York, pp 214–215

[PMS03] Paternò FM, Mori G, Santoro C (2003) Tool support for designing nomadic applications. In: Proceedings of 8th International Conference on Intelligent User Interfaces IUI'03. ACM, New York, pp 141–148

[PS02] Paternò F, Santoro C (2002) One model, many interfaces. In: Kolski C, Vanderdonckt J (eds) CADUI 2002, vol 3. Kluwer, Dordrecht, pp 143–154

[Ree05] Reeve G (2005) A refinement theory for μCharts. Ph.D. thesis, The University of Waikato

[USI] User interface extensible markup language. Homepage for the USIXML project: http://www.usixml.org/

[Wir71] Wirth N (1971) Program development by stepwise refinement. Commun ACM 14(4):221–227
[XIM] eXtensible Interface Markup Language. Homepage for the XIML project: http://www.ximl.org/

Received 30 April 2008
Accepted in revised form 6 October 2008 by A. Cerone, P. Curzon and D.A. Duce
Published online 19 November 2008