Formal Models and Refinement for Graphical User Interface Design

Judy Bowen

December 9, 2008

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. 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? User-centred design approaches, which incorporate many different techniques which we may consider as informal, seek to consider these issues so that the UIs we build are designed around the needs and capabilities of real users.

Both formal methods and user-centred design are important and beneficial in the development of underlying system functionality and user interfaces respectively. Given this we would like to be able to use both approaches in one integrated software development process. Their differences, however, make this a challenging objective. In this thesis we present a solution this problem by describing models and techniques which provide a bridge between the existing work of user-centred design practitioners and formal methods practitioners enabling us to incorporate (representations of) informal design artefacts into a formal software development process. We then use these models as the basis for a refinement theory for user interfaces which allows interface designers to retain their informal design methods whilst providing an underlying theory grounded in the trace refinement theory of the μCharts language.

Acknowledgements

My sincere thanks go to my supervisor Steve Reeves for his enthusiasm, patience and encouragement throughout this process and for teaching me valuable research skills. I could not have even contemplated undertaking this work without the support of my partner, Jane Stokes, and I thank her for believing in me and remaining positive through all of the ups and downs of my research. Thanks also to my co-supervisors Dave Nichols and Mark Utting for their input and feedback. I am grateful to the Computer Science department, the University of Waikato, and the New Zealand Federation of Graduate Women for providing financial support. Lastly I would like to thank friends and colleagues at the University of Waikato and elsewhere who have made this such an enjoyable undertaking.

Contents

Chapter 1

Introduction

This thesis presents an investigation into the integration of formal methods and user-centred design techniques within a software development process. Software development itself is a complex task. As new types of hardware, new modes of interaction and new contexts of use evolve this complexity increases. Frequently there are tensions between the high-level functional requirements for the system, the needs and desires of the users of the system, the expectations of the project managers running the development teams and the approach and understanding of the developers. In order to ensure we satisfy business, functional and user requirements we must consider, and resolve, all of these tensions. Ideally, we must find a development process which supports this resolution and which enhances the development experience.

Just as the types and uses of software continue to evolve, so too do the methods used in its development. Over the years various new approaches, including languages, types of languages, programming approaches *etc.* have been proposed, adopted and abandoned. These have resulted from shifts both within the business environments that software is being developed for, and a maturing of the disciplines of computer science and computer programming. While the methods and practical applications used have changed, the desire to build correct software has not, and there are, therefore, basic software development principles which remain consistent. One such principle is the use

of precise and rigorous methods used to plan, develop and test software prior to implementation. We will refer to such methods throughout this thesis as *formal methods*. Another principle is understanding that users are central to the software development process and that techniques and design approaches should be used which ensure that the needs of the user remain key. We will refer to these methods as *user-centred design*. There are, of course, additional underlying principles, but the focus of our research, and therefore of this thesis, is formal methods and user-centred design (UCD).

While there are many different possible methods of designing and engineering software systems, the choice of which of these to use may be driven by such things as the domain of the problem, the experience and preference of the designers, in-house guidelines, the type of software being developed and the end-users of the system. Given the size and scale of most (non-trivial) software applications it is common for the development to be split between several different groups of developers, each working on particular parts of that application. Within larger organisations these groups may work in separate physical locations, be led by different team leaders or project managers and have very little knowledge of the details of other parts of the project or even the project as a whole. Each group may adopt a different development approach depending on the part of the application they are working on. For example, network engineers concerned with security aspects of the physical hardware the software application will run on are likely to take a different approach than that of graphic designers who are responsible for visual elements of the user interface. All of these add further complexity to an already challenging process and create the additional burden of ensuring compatibility of goals and outputs across the various teams.

Regardless of the overall development approach taken, our stance is that it is desirable that certain key principles are adhered to. So, for example, we want to ensure certain properties hold true of our software before we implement it, and as such wish to take a strong software engineering approach to our design

and implementation, that is, make use of formal methods. At the same time we wish to consider the design of the software (and in particular the user interface to that software) from the perspective of the user, and ensure that the system we build is not only functionally correct, but also that it meets the requirements of those users. What we want, therefore, is an integrated approach to software development, and one which ensures that all of the functional requirements are correctly met and that the requirements and expectations of the users are similarly satisfied. We want to adopt both formal methods and user-centred design within a software development process.

1.1 Background

So far we have identified two underlying principles which we wish to focus on for our research into software development methods, namely formal methods and user-centred design. We will next discuss in more detail what we mean by these terms and outline our understanding of how they are used, and why they are useful, within software engineering.

Formal methods is a term used to describe any technique, or collection of techniques, which have a sound theoretical basis. That is, it refers to the use of notations and languages which are in some sense formal in that they have a mathematical basis, a logic (perhaps), and (possibly) a refinement theory. We use these languages and notations to create descriptions of the software before we build it, which we call specifications. Because the specification is in a language grounded in mathematical theory we are able to perform proofs upon the specification to ensure that our design really does what is intended. In fact there are two important functions to perform with the specification. The first is validation, making sure that the specification correctly describes what is intended (which is determined by the functional requirements for the software). The second is verification, making sure that the way we intend to satisfy that functionality is correct and that the software will be robust. That

is, we use formal methods to make sure that the software we intend to build is correct and will function properly in all circumstances.

There are many different languages and notations which come under the umbrella of formal methods. These may be mathematically-based description languages such as Z [2], B [5] and VDM [10] [3], or may be model-based such as the use of discrete-event systems [23] and model-based testing [100], or they may use process algebras such as CSP[49] and CCS [64]. This is by no means an exhaustive list, and just as new general methods for developing software continue to evolve, so too do new formal languages and notations. Whichever of these we select, our aim is clear: to correctly describe the software prior to implementation in order to ensure that we have correctly interpreted the requirements and plan to correctly implement them.

The process of creating a formal specification requires us to carefully consider the requirements of the system and understand how such requirements may be satisfied in some programming language (although we are often not concerned in these early stages what the final implementation language will be). The focus of the specification is on *what* the software does and we remain abstract about *how* this will be achieved.

Once we have created our specification and subsequently verified and validated it, we can move on to the next stage of the formal process, that of making the transformation from this abstract description to the concrete implementation. Within the formal process we refer to this as *refinement*. Generally we make the transition in small increments (*i.e.* we do not take a single step from specification to implementation, rather we move gradually towards the concrete implementation by becoming less and less abstract). Just as there are different formal languages, so too there are different refinement methods which are used as appropriate. The origins of refinement can be traced back to Dijkstra's work on giving meaning to programs via the weakest precondition and using this as the basis for a transformation method of program development [32]. Dijkstra's earlier work on program development via composition

[31] along with Wirth's research on the decomposition of refinement steps [103] led to the idea of stepwise refinement. There have been many notable developments and work in the intervening years on different aspects of refinement. Of these we consider Morgan's work on programming from specifications [65], Derrick and Boiten's work on refinement in Z [30], Woodcock and Davies' work on Z refinement and proof [104] and Henson and Reeves' work on developing a logic for the schema calculus of Z [46], [47] to be of particular interest within our own research, but this should in no way be considered an exhaustive list of references in this rich area of research.

When we consider the sorts of artefacts that are produced by formal methods practitioners, they are themselves formal. That is, they have some defined syntax and semantics (or we could not be certain of any of the requirements we have outlined), and possibly a logic associated with them, which allows us to be certain as to their meaning and how they may be manipulated. In order to understand the artefacts, however, you must understand the formal language used to describe them. They are intended to be used within a formal process for the purposes we have described rather as a general description of the software for a wider audience.

We use the languages and techniques of formal methods to consider the functionality of the software we are building from the perspective of what satisfies the requirements and how this will be achieved correctly. The 'how' is considered at an abstract level within the specification, so it describes high-level solutions rather than actual (low-level) code implementations. The description we have given above is the definition of formal methods we rely on throughout this thesis.

User-centred design (UCD) encompasses a range of different methods, techniques and approaches which have in common the fact that they involve users. When we talk about users we mean the end-users of the software we are developing, *i.e.* the real people who on a day-to-day basis will interact with the software (rather than perhaps their managers who may be responsible for

deciding upon the initial requirements of the software).

Commonly used UCD techniques include things like performing ethnographic studies, task analysis, prototyping, story-boarding, use of personas and scenarios, usability testing, and a host of others. Because these are general, commonly-used activities there is no single reference for any one of them, however descriptions of such techniques can be found in any number of design texts devoted to UCD, such as, for example [86], [43], [87] and [33]. UCD methods are structured activities which allow designers to fully understand the needs of the users, to communicate their ideas to those users and to integrate feedback from the users at all stages of the design process.

We refer to UCD techniques as informal, not because the activities have no structure or underlying methodology, but rather because the artefacts produced by such techniques are themselves informal. That is they do not have a defined meaning independent of the artefact which can be examined using the formal techniques we have described above, rather they provide an intuitive understanding of what is intended. This is deliberate since the purpose of employing such techniques is to maintain communication with users so that they become involved in the design process. Therefore the artefacts used as the basis for that communication must support this. That is, they should be easily understood by the users and not require knowledge of programming languages or complex notations or even the extended requirements of software development beyond the purpose of the actual software being developed. In addition, the artefacts often support collaboration and direct input from the user. Lo-fidelity artefacts, such as paper prototypes, encourage the user to make suggestions and changes because their nature allows them to appear as if there is not much invested in them by way of time and care from the designer (although this is usually not true). Subsequently we may develop more hi-fidelity artefacts, such as computer-based prototypes, which allow the users to get a better idea of the real look and feel of the design, which in turn generates different types of feedback. We expect that throughout a UCD process

a number of different artefacts will be produced (often starting with lo-fidelity and ending up with hi-fidelity artefacts) which support user understanding and encourage collaboration.

We consider UCD an important part of our aim of developing correct software because the inability of users to interact correctly with software can be just as problematic as software which is itself not correct.

We have discussed formal methods and user-centred design as two different possible approaches to developing software. We have given a brief description of what we mean by these terms in this chapter, and in chapter 2 we will expand on this and give detailed descriptions of both and how we will use them throughout the rest of this thesis. We have chosen these two approaches because they offer a pragmatic approach to the problem of correctly designing and implementing usable software. Both have their own history of use and development. They also both refer to a collection of possible methods and techniques which fit under the respective umbrella of either formal methods or UCD rather than being one single activity. Between them, they cover most of the things we are concerned about in our software development process. That is, they include correctly building underlying system functionality as well as developing systems that are usable. The intentions of both formal methods and UCD are in one sense the same, to build software that is correct and which behaves in accordance with requirements. The major difference is that with formal methods the focus is on the functional requirements, whereas with UCD the focus is on the user and how they will interact with the software in order to satisfy their requirements.

We have stated that we would like to use these two approaches together within one development process. However, this is not straightforward and leads to a number of potential difficulties, we discuss these next.

1.2 Problem Identification

As we have discussed, formal methods and UCD techniques both work successfully as approaches to software development. Formal methods are important for considering the functionality of a system, whereas UCD techniques ensure that the software meets the needs of the users and in particular that the user interface (UI) to the system is usable.

Because the focus of these methods is so different it is often the case that design teams will employ one or the other. Experts in formal methods are generally not experts in interface design and the mathematical basis of the languages they use makes it difficult for them to interact with system users. Similarly, usability experts and interface designers are unlikely to be overly familiar with formal methods; the sorts of artefacts they are working with are designed to be simple and understandable rather than robust and precise. As we have stated there is often a separation of concerns within the design environment which means that the underlying application logic can be developed using formal methods, and the interface to the system can be designed following a UCD approach.

Separation of the design and implementation of the UI of a system from what we will refer to as the underlying system behaviour (or application logic) is often seen as a pragmatic approach. The development of user interface management systems (UIMS), as exemplified by the Seeheim model [77], was driven, in part, by an understanding of the different approaches required for these two things (as well as a desire for portability of user interfaces). Separation allows us to focus on the different concerns presented by the different parts of the system, as well as enabling use of different techniques as described above. An examination of UIMS and interface architectures is given in [39] where Evers discusses how their use is motivated by the desire for:

> "adaptability, portability, complexity handling and separation of concerns of interactive software."

In this work he also describes how such a separation can lead to adaptability problems. Whilst Evers' concerns are primarily related to semantic feedback between interface and application logic his comments highlight one of the problems caused by the separation. We discuss this, as well as another of the problems of such separation, next.

Assuming we wish to use both formal methods and UCD (which we have stated is our aim) then we need to further consider the impact of following the separation route described above. There are two major problems with this approach. The first problem is that by following two separate design streams we risk each part of the process ignoring important considerations of the other (of which they have no knowledge). Whilst formal methods are well-suited to the functional requirements of the underlying system, they are weak on usability and user issues. Indeed, such issues are unlikely to be immediately obvious from functional requirements. Similarly, the focus on users required for the UCD process makes it harder to see the 'big-picture', by which we mean the overall context of the software. This may include important requirements which, because they are not directly obvious to users, are easily overlooked within the UCD process (such as underlying security requirements for example). This in turn leads to an incompatibility between the goals of the separate design teams. There is no guarantee that they each have the same understanding of the overall implementation, which may lead to each side making design choices which are in conflict with the other. This problem has been noted many times previously, as Harrison and Loer [61] point out:

> "...perspectives and practices (of system engineers and usability engineers) are different and are often performed separately.."

The second problem with separating the functional core from the UI during the design is that at some point we must, of course, bring the two parts together in one final implementation. The purpose of following a formal methods approach is to be sure that our final implementation has been designed and implemented correctly, and we perform proofs and tests along the way

to guarantee that this is so. However, no such guarantees or proofs exist for the UI design, so the overall combination of system and UI cannot itself be guaranteed to satisfy the properties we have previously proven. From a formal methods perspective this is not satisfactory.

In order to use formal methods and UCD together, we must find another approach, one which allows us to integrate the two. The differences between the two approaches, and indeed between the artefacts they produce, make it unclear how this may be achieved. We might end up watering down the formality to make it understandable for users (but still, therefore, unsatisfactory from the formal methods perspective) or we could make our UCD designs more complex and include them in the formal notation of the specification (which is unsatisfactory both for UI designers and users, who are unlikely to fully understand such specifications). How then can we successfully integrate these in a way which has neither of these problems, but which allows (potentially) separate design teams to use different methods and be sure that their goal is the same? This is the problem we set out to address, and solve, in this thesis.

1.3 Thesis Statement and Research Questions

Based on our desire to use formal methods and UCD together in a single design process, and the problems we have identified above, we now state the intentions of this thesis.

> **Hypothesis**
>
> Formal software development methods and informal user-centred design methods for interface development can be used together in an integrated and useful manner. This will enable a suitably designed user interface to be constructed, along with the underlying system functionality, within a formal software development process.

Our hypothesis generates the following research questions:

- How can we integrate the formal artefacts of a formal methods process with the informal artefacts of a UCD process?
  - In particular, can we find a way to usefully formalise the informal?
- What would such formalisms look like?
- What effect does such integration have on refinement?
- What does refinement mean in the context of user interface design?
- How can we describe the composition of both system and UI formally?
- Is there a single refinement process which can be used for such a composition?
- If we refine either part of the composition independently does this imply a refinement for the composition?
- What are the benefits we obtain from this integration?

These are the questions that this thesis will answer.

1.4 Summary

The focus of our research is in exploring ways of integrating formal methods and user-centred design. In particular, we will show how we can take existing informal design artefacts (such as those typically produced by UCD methods) and integrate them into a formal software development process. This will include: looking at ways of creating formal models of UIs and UI designs; using such models to prove properties such as consistency between specifications and UI designs, as well as properties about the designs themselves; consideration of refinement for systems and UIs; and monotonicity of such refinement.

This thesis provides the following new contributions:

- A formal model for describing informal design artefacts (the *presentation model*);

- An extension to presentation models describing the dynamic UI behaviour encapsulated in their designs (*presentation interaction models*, or *PIMs*);

- Integration of UI design models and formal software development methods using the presentation models and PIMs in conjunction with formal specifications;

- A description of what characterizes UI refinement;

- A monotonic refinement theory for system and UI compositions using the μCharts language [84], [79];

1.5 Thesis Structure

In the next chapter of this thesis we will describe the context of our research. We begin by examining related work and different approaches that have been taken to the problem we have described. We next extend this to show how our approach to solving this problem is different from existing research and explain why we have taken this different approach and the benefits it provides. We revisit the general ideas and concepts of formal methods, user-centred design and refinement as they will be treated in this thesis in order to provide a platform for our subsequent chapters. Finally we give a brief description of some real-world design practices and case-studies to show the application of our work and how it fits into real-world software development. We conclude with a summary of the background and context of our research.

In chapter 3 we introduce some examples which will be used throughout the thesis as the basis for explaining our methods. We give a description of the requirements for each of the pieces of software (including some variants)

and describe some of the design artefacts which are generated by these requirements.

In chapter 4 we introduce the models we have developed for UI designs [15]. We describe their syntax and semantics and explain their uses [14], [18]. We discuss the benefits of using such models and reflect on our experience of practical applications of the models [17].

Chapter 5 discusses the problems relating to bringing together the formal descriptions of the underlying system and the UI. We describe why this is both necessary and important, and describe conditions upon the composition which ensure that we model compatible system and UI pairs which then describe a valid system. We give a brief description of the μCharts language and show how we can use it to model the composition of systems and UIs and how this captures the validity conditions previously described.

Chapter 6 describes refinement considerations. We begin by explaining why refinement is important in our research and what our aims are. We next present some general notions of refinement and use these as the basis for developing a description of refinement for UIs [16]. We then consider how we can describe this formally using the μCharts language. We show how monotonicity of refinement is captured by μCharts' refinement theory in conjunction with our validity conditions. Next, we discuss different types of refinement and their uses. We show how our formal refinement model both supports and enhances the intuitive refinement that typically takes place during an iterative UI design process. We end the chapter with a discussion of the benefits of our refinement process and future possibilities for this work.

In chapter 7 we give some examples of using the refinement techniques we have developed on larger problems. We also introduce the concept of vertical refinement and show, by example, how this differs from our previous examples.

Finally, in chapter 8, we present an overview of our work and show what has been achieved. We discuss possible future work before presenting our final conclusions and outlining how we have met the initial aims of this thesis.

Chapter 2

Context

2.1 Literature Review

Research into the application of formal methods for user interface and interactive system design has a long history. The subject can be, and has been, approached from two different sides. On the one hand formal methods practitioners have tried to find ways to include user interfaces, user concerns and interactivity between system and UI into their formalisms. An example of this approach can be seen in the formal methods text "The Way of Z"[51] where Jacky gives a description of the graphical user interface of a console to a radiation machine as a Z specification. On the other hand, user interface designers and UCD practitioners have tried to find ways of formalising parts of the design process so that they are able to do the sorts of things normally reserved for formal methods practitioners, namely find guarantees of correctness within parts of their work. Often, such research will focus on one specific area of concern, such as security, Johnson's work of the 1990's on safety critical interactive systems being a good example of such an approach [54].

The research presented in this chapter is intended to provide context in our specific area of interest, that of finding ways to integrate formal methods and UCD techniques. As such, it is not intended to be a fully comprehensive overview of *all* work in these areas, but rather specific examples designed to

illustrate particular approaches which are relevant to our research. Collections of essays and research papers on the subject of formal methods and user interface design such as [45] and [69] and overviews of this research area such as [44] provide additional background beyond the work we describe, although we will also refer to specific examples from these collections as relevant within the rest of this chapter.

In order to usefully describe the existing research and compare it with our own approach, we group it into several categories. These categories are useful in that they describe the main approaches taken to the research which helps to make clear the underlying similarities between some of the approaches. However, inclusion in the same category does not mean that two pieces of research necessarily have the same theoretical basis, just that they are similar in their underlying intentions. Similarly, the categorisation is based on our interpretation of the research rather than necessarily being that of the authors. We will next describe each of the categories and give examples of research these areas in order to discuss both the strengths, and weaknesses, of each approach in terms of developing a unified software development process.

2.1.1 Including the Interface in the Formal Specification

One obvious way to ensure that interface concerns are considered formally is for formal methods practitioners to include a description of the interface in the initial system specification. This removes the problems that separation of UI and underlying functionality can cause by aiming for a single design process based upon a formal specification which includes aspects of interactive behaviour and interface concerns. This is an approach taken in some of the earliest research in this area. Sufrin [91] states that:

> "Designing an effective user interface to a complex information system is difficult, since it cannot be done in isolation from the design of the information system itself."

To illustrate this point and show a possible solution he uses the example of a mail system and models both the underlying system and the interface concerns using Z. In fact, the interface specification amounts to a description of the operations required by the user and so extends a specification based on functional requirements to ensure that user requirements are also included. The result is a more complete specification of the system which allows formal considerations of user operations, but without any consideration of implementation of the interface, or how such operations will be made available (and understandable) to the users. For example he defines the notion of each user having a virtual desk and describes what observations these users have of this using the following Z schema:

$$
\begin{array}{|l}
\hline
DESK \\
clock : T \\
owner : P \\
in, out : \mathbb{F}\, MESSAGE \\
\hline
\forall\, message : out \bullet message.source = owner \\
\forall\, message : in \bullet message.dest = owner \\
\forall\, message : (in \cup out) \bullet message.postmark \leq clock \\
\forall\, m_1, m_2 : (in \cup out) \bullet (m_1.postmark = m_2.postmark) \land \\
\qquad (m_1.source = m2 > source) \Rightarrow m_1 = m_2 \\
\hline
\end{array}
$$

While this constrains which messages are visible to users of the system (and so specifies the required security aspects of the UI) it does not provide any information about how the interface to such a system might appear visually. In addition, the use of Z and quantified statements in the schemas to describe the constraints on which messages should be visible in the UI are not necessarily useful to those responsible for the UI design who are unlikely to be familiar with the notation.

A variation of this approach is seen in Chubb's work [24] which takes the approach of explicitly modelling both windowing and event handling using Z and state transition diagrams. The result is a low level representation of user interactions (by way of type and position of mouse clicks) to determine whether they invoke a specific command or not. The overall aim of the work is to show that extending a specification to include such observations of an interface is a satisfactory refinement of the initial specification, that is, the total system is correct based on the initial functional specification. The goal is to show that a suitably designed interface does not change, or restrict, the specified behaviour of the system in any way. The consideration of the UI is at a behavioural level where user interactions are described as operations, with conditions on how and when such interactions may occur. As such, the specification does not give any indication as to how a suitable design might be inferred from the formalism, or what the design implications are of the interaction requirements.

Also within this category is one of the most influential areas of research, that of *Interactors*. We claim these are influential because since their development in the early 1990s they have formed the basis for much of the research which follows, and they are still in use (in many different forms) today. Interactors were originally developed at York by Harrison and Duke [36] based on the earlier work of Paterno and Faconti [40]. In [37] Duke and Harrison describe the background to their work as:

> "... the problem of relating different representations of the design artefact, each potentially represented in a distinct formalism"

and propose interactors as a possible solution. They combine different viewpoints of the system development (functional, interaction and presentation issues) into a single formal representation, the interactor. Interactors allow the specification of an interactive system by way of components (or interactors) which have some internal state representing functionality, a perceivable state representing something that is somehow made visible to users, and sets of

actions which bring about changes in both of these states. Interactors are language independent and have been described using Z [20], ObjectZ [50], VDM [35] and Lotos [75] among others. The original structure of interactors has also been extended in several different ways to include, for example, the ability to consider, and record, design decisions [19].

Capturing system behaviour, interactive behaviour and presentation issues into a single component in this way allows the interface of a system and the underlying functionality to be described together in a single specification. The representation of presentation issues, however, is not one which can easily be integrated into the design process typically undertaken by interface designers, where visual representations of actual UI appearance are important. Similarly, the use of formal notations such as Z or Lotos to describe the interactors mean that they cannot be used as the basis for communication with users or be easily interpreted by designers. Their use aims to ensure that all parts of the system are considered equally and form part of a specification, but they are rooted in the formal aspects of design.

Whilst Interactors remain popular as the basis for many methods within the community interested in formalisms for interface designs they are not similarly well used within the UCD and UI design communities. One possible reason for this is that this form of abstraction away from the graphical nature of the UI leaves open the question of how the UI will be visually presented, instead considering presentation by way of a mapping from interactors to presentation concepts, which is not an intuitive approach for UI designers.

An extension to the use of interactors, proposed by Bramwell [19], tries to solve some of these issues by using the formalism to support an iterative design approach. Rather than using formal methods to try and describe interactive systems separate from their design and implementation he instead tries to develop:

> "practical formal methods for developing interactive systems."

He outlines how user requirements may be specific, and not reflected in initial requirements of a specification. As such he introduces design choices as a set of possible refinement strands from an initial specification and uses interactors as the basis for both recording, and reasoning about, the choices made in light of such new requirements. He develops a new version of interactors to support this work (using action systems in conjunction with CSP). The work goes some way towards tackling the problem of incompatibilities between system requirements and user concerns which may arise during the UI design process, but because it still uses the approach of bringing the UI into the formal specification rather than considering a separate, more user-focused, UI design approach it is only a partial solution. Once again the formal descriptions used to specify parts of the interface are well-suited to the requirements of specification and proof but less satisfactory to user interface design. For example a dialogue for a menu hierarchy used to load a file is given in CSP as:

$$\alpha(LOAD_1) = selmenu, selload, input.x : Text, load.x : Text$$
$$LOAD_1 \mathrel{\widehat{=}} selmenu \rightarrow selload \rightarrow input?x \rightarrow load!x \rightarrow LOAD_1$$

which allows the actions and performance of the described events to be considered formally. However, a more typical UCD artefact for describing a similar dialogue might be a prototype or storyboard such as that given in figure 2.1.

These then are some of the examples of research which attempt (in various ways) to consider the interface and the user as part of a formal specification. The general aim is of ensuring that the total system being built has the guarantees of correctness we normally expect when using formal methods, and that it has been developed with a single, clearly defined goal. The benefits of such work are that it leads to a more complete specification where the needs and requirements of users are considered at the same stage as the functional requirements. Also the interaction between system and UI is defined at the specification stage which goes some way to ensuring there are not problems

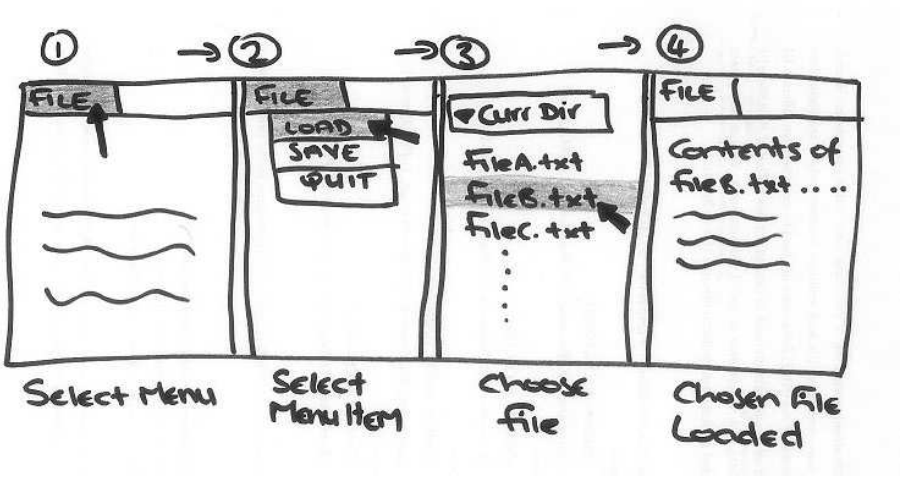


Figure 2.1: UI Description for File Dialogue

of integration at the implementation stage. Such an approach undoubtedly leads to the development of systems which better support the UI functionality (as they have been considered together with the functional aspects) as well as enabling the use of proof techniques to consider the interactive behaviours of the system.

The weaknesses of this approach are that they make the development of the interface a formal process by including it as part of the formal specification. This has the effect of producing artefacts which cannot be used as a basis for communication with users, and therefore either contradicts taking a UCD approach to the UI design, or creates a gulf between the visual design of the UI and the interaction considerations. The proofs of behavioural properties are still, therefore, weakened at implementation time as there is no way of linking the two parts of the design at this level. By restricting descriptions of the interface to a formal language we take away the ability of UI designers and users to communicate around the design. In human-centred approaches the design of UIs is increasingly seen as an artistic discipline as much as a software development one, and as such designers are unlikely to want to change their approach from one that is primarily visual to one which is primarily formal.

2.1.2 Creating Models of the Interface

A slightly different approach to the problem can be taken which involves creating formal models of the interface as separate entities from the system specification. The intention here is to create a model of the interface and interactive elements which allows us to treat them formally and so get the benefits of formal methods in general by performing proofs to check for desirable properties and correctness.

One of the earliest examples of this approach is the PIE model. This was initially described by Dix and Runciman [34] and subsequently extended in several ways. The model describes interactive systems by way of an abstraction of programs (P) which consist of sequences of commands, effects of inputs (E) and an interpretation function between these two (I). The use of these triples allows analysis of certain properties of the interface such as observability, reachability, undo *etc.* The model creates an abstraction of the interface which is not connected to the underlying system functionality, but which is also not connected to a more concrete understanding of either visual elements of a UI, or the users of the system (beyond the pre-determined inputs). In this way the PIE model retains the separation between interface and application logic which we have described earlier. While it does allow a more formal treatment of the interface itself, it provides no guarantees about compatibility with the rest of the system or ways to link the model to a system specification.

A different approach to modelling is that taken by Thimbleby *et al.* on using matrix algebra and Markov models to describe UIs [96], [94]. This work considers the usability of the interface without any concept of the design or appearance of that UI (*i.e.* is not concerned with how, or why, users behave when interacting with an interface) but rather considers only the possibilities of interaction in terms of the underlying possible state changes and their possibilities (or probabilities in the case of the work on Markov models). In contrast, our work relies on a UCD approach to the design which has the opposite goal in that it considers the users and the visual designs as the starting

point.

Some proposed models take a less abstract approach and do try to create a more obvious link between visual elements of a UI and models of that UI and its behaviours. In [21] Bumbulis *et al.* propose a component-based model which takes an iterative approach to the design process and considers models of prototypes and user feedback. They create a UI specification which can be used to construct either a prototype for user testing or a model for formal reasoning, and therefore provide a link between the formality and a UCD approach. The language they propose (IL) is, however, a very low level description (almost at the level of code). For example a visual component on a UI which contains both a slider and a dial is described by the following (where the slider and dial have both been previously declared in a similar manner):

$$
\begin{aligned}
&\mathit{Frame\ primitive}\\
&\mathit{Dial\ changed} > \mathit{set} < \mathit{primitive}\\
&\mathit{Slider\ changed} > \mathit{set} < \mathit{primitive}\\
&\mathit{Main}\{\\
&\qquad f : \mathit{Frame}\, f.d : \mathit{Dial}\, f.s : \mathit{Slider}\\
&\qquad f.d.\mathit{changed} \rightarrow f.s.\mathit{set}\\
&\qquad f.s.\mathit{changed} \rightarrow f.d.\mathit{set}\\
&\}
\end{aligned}
$$

and it is this which in turn drives the creation of visual components. In this example a direct translation from IL to the dynamic interface language Tcl/Tk [92] is used to create components for the interface rather than visual designs forming the basis for their models.

Another benefit of creating models of the interface is that it enables the use of model checking. As we have discussed it is often the case that creating formal models of the interface is not helpful for interface designers (who are not familiar with such formalisms), and with this in mind Loer and Harrison undertook to create a framework for model checking (IFADIS) [59], [60] and [61], which would make the task easier for usability experts. They aim to use model checking as a way of analysing systems for dependability and performing usability inspections (more usually undertaken by real users interacting with the software). Despite their intention to make the process of model-checking more palatable for usability engineers they admit that the work does not go far enough in this respect, and that:

> "while the toolset would be of value to the broader community of system engineers not familiar with model checking, it continued to be problematic for use by usability engineers" [61]

They identify one of the major causes of this problem as being:

> "..in practice there is a gap between the artefacts used by design teams ... and the inputs required by model-checking tools" [61]

Once more this work abstracts away from any visual notions of the interface, and user concerns are considered in terms of interaction requirements and possibilities rather than from considerations of prototypes.

A more recent development in UI modelling is that of creating abstractions of UIs for portability and plasticity. It has become increasingly common for software to be developed to run on multiple platforms (*e.g.* desktop machines, PDAs, mobile phones, embedded devices *etc.*) as well as different operating systems. This has led to a desire for portability - developing a UI which can

easily be adapted between different platforms, as well as plasticity - the ability of a UI to adapt based on its context (which may include current hardware, user location *etc.*) There are a number of different research groups using XML-like languages to model UIs for these purposes. One such example is the XIML project [78], [106] whose aim is to develop an XML-based language to describe interaction data and thus provide an abstract model of the UI.

Paterno *et al.* have developed a tool, TERESA, which allows models at varying levels of abstraction to be created and for the transformation of task models to abstract UI models, which can then be adapted for different platforms [73], [74]. The tool uses an XML task model, based on ConcurTaskTrees [71] to generate an XML abstract UI model which can then be used to automatically generate different concrete UIs (using XHTML). Because there is a process of moving from an abstract model to a concrete UI there are closer links between visual aspects of the design and the formal model. However, once again the process is driven from the formal side (so the model leads to the UI design) rather than vice versa.

A third variant of the XML approach uses a language called USIXML [99],[58]. The emphasis in this research is on providing a translation mechanism between different abstract models of UIs which allows both a transformation between different versions of the same UI, as well as development of concrete UIs from these models. As part of this project a number of tools have been developed to support UI design (*e.g.* GrafiXML, SketchiXML) and it is also compatible with the TERESA tool of the XIML group. The development of such tools is an important step as it provides design environments similar to commonly used tools, such as Visual Studio [63], where designers can quickly create computer-based prototypes using drag and drop techniques.

Because the purpose of these models is to support abstract UI models which can be altered for different contexts, they are deliberately separate from any model of the underlying system. They aim to describe the UI and its interactions at a level which enables them to subsequently be associated with

the relevant system model (depending on platform or context). A central aim of all of this work is ensuring that usability is maintained across the various platforms, however this is defined by the underlying task models rather than through user interaction.

The use of XML-based languages to describe portable UIs is not limited to UI modelling approaches. XML is used as the underlying basis for both XAML (Microsoft's Extensible Application Markup Language [105]) and XUL (Mozilla's XML User interface Language [107]). Both XAML and XUL separate the description of the UI from the rest of the application and provide descriptions which can be used to create portable UIs, for example to create web-pages or the front-end to applications on the fly. The UI descriptions capture detailed information about the types of components and widgets used (XAML includes information about visual appearance details such as size, colour, font *etc.*) but do not include information about functionality. XAML uses event-management to link the UI description to the functional code of the application while XUL requires separate implementation of functionality which is then linked to the UI via signals and signal handlers. For example a XUL toggle button has a "toggle" signal which must then be linked to a signal handler which in turn is linked to the code implementing the button's behaviour. While both XAML and XUL are useful for UI portability they are declarative descriptions of the UI rather than models which can be used to analyse UI behaviour.

Creating UI models independently of system specifications provides a number of benefits. Having a formalisation of the UI allows us to analyse it and check for particular properties in the same way the use of formal methods for application logic does. In addition such models are a useful basis for the transformation approach, described above, when developing UIs for multiple platforms or contexts. The main problem with such models is that they assume a new way of working for interface designers. They require a particular language to be learned and a new set of tools to be adopted rather than provid-

ing a bridge between existing design techniques and the modelling techniques. Also, retaining the separation of UI and system and the use of UI modelling languages which are distinct from the formalities used for the application logic means that we are left with the problems of ensuring compatibility and of preserving proven properties at the time of integration.

2.1.3 Formalising Parts of the Interface Design Process

The third category of research we describe is that of formalising individual parts of the design process. This category encompasses research which is developing techniques for applying formal methods within one part of the overall process of developing the UI. Unlike the previous categories this work is less driven by formal methods practitioners and more from within the UI design community.

Undoubtedly the area which has benefited most from this type of research relates to task analysis. Task analysis is an early design process which involves taking the user requirements and structuring them in some way (perhaps into hierarchical tasks) so that they can then be used as the basis for the UI design. We would expect that some sort of task analysis would take place in any UCD approach as it is one of the key parts of ensuring user requirements are satisfactorily met.

In [72] Paterno suggests modelling user tasks in UML based on the ConcurTaskTrees (CTT) [71] notation and using this as a basis for describing interactions. UML is chosen because it is already widely used and there is, therefore, likely to be some industry support for it. Paterno does point out, however, that UML already has nine different notations (at the time of writing, there are currently thirteen), most of which are not used in practice by software engineers and that adding another one may not prove successful. In our previous section we described the use of UI models, and many of these have task analysis as their basis using either CTT, task action grammars (TAG) [76] or hierarchical task analysis [85]. Each of these can themselves be considered

a formal method allowing for some formal analysis of user requirements.

Kaindl and Jezek examine a method of assisting and recording design decisions by transforming task analysis into widget selection decisions [55]. They do this by transforming standard task analysis data into a series of increasingly detailed task hierarchies which are then classified by types depending on whether they are user or system actions and how they inter-relate to the other tasks in the classification. Their widget hierarchy has similarities to the one we proposed in [13] which we will see in chapter 4, in that it classifies widgets in terms of behaviours. Their purpose is to support novice UI designers by assisting with design decisions based on their hierarchy. As such it provides a formal method for supporting design by helping the instantiation of widgets from tasks.

Another commonly used UCD approach is the use of scenarios. These are proposed situations which reflect real things which a user might wish to do with a piece of software and considerations of how they might go about doing it. As such they extend the tasks which have been defined and develop a premise like "what if a person like *A* wanted to perform a task like *B*?". Poll *et al.* [101] research links between developing scenarios to the development of use cases, and propose a method for combining these use case maps (as they have called these scenario-based descriptions) and formal methods to validate scenarios against user requirements. This allows analysis of consistency and completeness by defining how users can interact (how they move between tasks). It is, therefore, similar to much of the work on task analysis in that it formalises a design activity to enable some sort of analysis of the artefacts produced.

Our final examples in this category are the use of formal techniques for the purposes of interface testing. Testing is an important part of the UCD process and begins to take place as soon as initial designs are developed (when prototypes are used to give the users an idea of early design decisions) and continues right through until after the application has been implemented (when

usability testing by both users and experts is performed). Usability testing is seen as an important, but often costly, and lengthy, process. For this reason the ability to perform some of this testing formally (and without users) is appealing.

Some of the research in this area attempts to find ways of using existing formalisms to analyse UI models. Belli [9] uses finite-state automata to consider interaction sequences and generate models of both correct and incorrect interaction to determine where errors may occur. A similar approach is taken by Rosis and Pizzutilo using Petri nets to both model the UI and evaluate its usability and correctness [28]. Formal methods can also enable the automatic generation of tests for UIs. Paiva *et al.* use a combination of interactors and VDM for this purpose [67]. Paiva subsequently extends this work by reverse-engineering applications written in C# (using Microsoft's Spec# tool [89]) and using finite-state machines to generate test cases.

All of these examples show how particular parts of the UI design process can be made more formal by developing individual techniques. They therefore enable us to get the benefit of the use of formal methods within informal processes. The main problem with these approaches is that because they focus on just one part of the design process they do not provide a total solution to the problem of integrating UI design with a formal software development approach. In particular, each proposes a different formalism, or uses existing formalisms in different ways, making it difficult to try and combine them into a more unified approach.

2.1.4 Formal Models of Users and User Cognition

The final category we describe is that of modelling the users. Such an approach provides the ability to consider the UI in tandem with particular models of user behaviours and/or cognition with the intention of finding mismatches and potential conflicts. Trying to capture possible human behaviours and human understanding of a system within a formal model is an enormous task,

and therefore the research in this area often focuses on particular aspects of behaviour and understanding which are bounded (rather than trying to model *all* possibilities).

One approach is to consider the possible errors a user may make, and then use the probability that such an error might be made as a metric in quantifying system safety. In his work on safety-critical system design Johnson takes this approach and uses it in conjunction with a probabilistic logic to model both temporal properties of user interaction and error probabilities in order to assess risk [53]. In this example it is the potential for human error which is being modelled rather than actual user behaviours. The result is that designs can be considered in terms of how likely they are to cause human error, and therefore fail. As such this method provides a way of supporting the development of more robust UIs.

In [25] Curzon and Blandford use formal cognitive models of user actions, which they define as rational actions a user will take to achieve a goal. These are then used to derive design rules. This is done to try and identify when it is correct to use particular design rules and in particular which rule to use when there are contradictory possibilities. The method is concerned with generic user models and examining how such users may make mistakes in particular interfaces and how particular design rules would prevent this. The user model in this work is based on behaviours and contexts, and principles of cognition are developed in order to model plausible behaviour. This research is linked to that of programmable user models (PUM) [22], [11] and [12], which consider what knowledge is required by a user in order to successfully interact with a particular interface. This work provides a strong link between design rationale and decision making and the potential for error based on user knowledge.

An extension to this work is seen in [4] where the authors extend the model of user knowledge to include context of use issues. The premise is that by including contextual concerns the models can be made richer and the interaction design improved.

One of the benefits we derive from this area of research is that it makes the user an important part of the development process by developing formal models of possible behaviours and using these as part of the basis for the design process. As such it seems similar to the UCD requirement of keeping the user central to the design process. However, the models are developed, in general, not from analysis of actual users of the system (primarily because this would be a huge task more suited to the area of psychology than that of computer science) but rather from the viewpoint of what is possible in terms of interactions and errors and how these may be prompted or avoided by particular design choices. One contrast to this is the work of John *et al.* [52] which aims to produce predictive models of cognition more rapidly than other approaches and which are more accurate. These models are developed by capturing of information from real users using prototyping.

2.1.5 Comments

We summarise the work we have discussed and where it fits into our categorisation in table 1, with the following key:

1 - Including UIs in formal specifications

2 - Creating models of the interface

3 - Formalising parts of the UI design process

4 - Formal models of users and user cognition

	1	2	3	4
Radiation machine GUI in Z [51]	√			
Mail system in Z [91]	√			
Data refinement models for UIs and systems [24]	√			
Interactors [36], [37], [20], [50], [35], [75], [19], [67]	√		√	
PIE [34]		√		
Safety critical systems [54]			√	√
Component-based models [21]		√		
IFADIS [59], [60], [61]		√	√	
XIML and USIXML [78], [106], [99], [58]		√	√	
CTT [71], [72]		√	√	
PUMs [22], [11], [4]				√
Modelling cognitive actions [25]				√
TERESA [73], [74]		√	√	
TAG [76]			√	
HTA [85]			√	
Task analysis for widget selection [55]			√	
Scenarios and use-cases [101]	√		√	
FSA for usability testing [9]			√	
Petri nets for usability testing [28] [9]			√	

Table 1

The categories we have described above provide a general description of the types of research in the area of formal methods and user interface design, along with specific examples. Sometimes the categorisation is not as clear cut as we have made it seem as several different approaches above may be combined, for example the use of task models (such as CTT) as a basis for models of the interface which can be used to describe nomadic applications.

Each of the different categories provides its own benefits, which we have described, but overall the sorts of benefits we see are: being able to provide guarantees of correctness of aspects of UI design; a more thorough system specification (in that it includes UI concerns); and a tighter link between the requirements of UIs and underlying system at an early stage.

We have also seen that there are some weaknesses in the various approaches in terms of developing a unified approach to software development. Of course, many of these weaknesses relate to our desire for integration rather than being a weakness of the research itself in terms of what it set out to achieve. The point is that the research we have described is not a suitable solution for our problem, rather than the research being unsuitable or unsuccessful in general.

Bringing the design of the UI into a formal process (by, for example, including the UI in system specifications) has advantages in that it provides a single focus for the design, and may therefore seem to help ensure that all parts of the design have the same goal. However, as we have discussed, this is not the most appropriate approach for UI designers, particularly when following a UCD approach. The languages and notations of such formal specifications do not lend themselves to easy communication with users and are unlikely to be easily adopted by designers concerned with visual elements, as well as behavioural elements, of the UI design.

We can avoid some of these problems by developing new ways of modelling UIs, which may allow us to create formalisms which are more light-weight and easier to adopt for non-formal experts. However, the danger here is that the formalisms we use are not strong enough to satisfy formal methods practition-

ers and that we end up developing many new languages and notations which are not compatible with existing methods. The research which uses existing languages (such as Z or Object Z) as the basis for UI models avoids this problem, but such work tends to fall into the category of approaching UI design from a formal perspective, and therefore inherits those problems described above.

We can realise some of the benefits we would normally expect from a formal approach by applying formality to particular parts of interface design. This enables us to produce more formal artefacts within our design process and means that even when we take a UCD approach to UI design we are still able to apply formality in some places and get the benefits associated with it. However, in terms of the overall application development, if we are applying formality only to certain parts of the UI design we still have the problem of not being able to guarantee that the final implementation meets all of the requirements and so we cannot state that properties we have proved of parts of the design hold true for the complete application.

The practice of modelling users by way of cognition, actions or error possibilities allows us to treat the user formally and keeps the user central to the formal development process, which is appealing from a UCD perspective. However, most of the work in this area develops general models of users and uses these models to consider requirements as opposed to UCD which works with real users.

2.1.6 Points of Difference

Now that we have an understanding of research in the area of formal methods and UI design and have considered some of the strengths and weaknesses of such research, we can examine why it is not suitable for what we are trying to achieve and outline how our approach differs from the work described.

A common theme amongst much of the existing research is that it abstracts away from any visual notion of the UI (other than abstract ideas of presenta-

tion or visualisation) and develops ways of modelling or describing UIs which are driven from a formal approach to the software development. Given that abstraction is often one of the key purposes of specification (by describing the problem without the context of implementation we have a clearer view of the problem) this may seem a reasonable approach, however, it is important to note that UI designers and UCD experts are often visual people (as concerned with the aesthetics of the design as they are with the functionality). They use particular methods and techniques because they work for them and are suitable for communicating with users during the design process. That is, driving the design from the user requirements and understanding of user capabilities as well as providing visual prototypes along the way as part of the design process is a suitable approach from the perspective of UCD designers. Within our research, we must then recognise the effectiveness of these methods and retain them. Regardless of how we solve the problem of integrating system formality with UI design we wish to use existing UI design techniques as our basis, rather than expecting UI designers to change their approach.

Just as we recognise the need to allow UI developers to continue using the methods they find most appropriate and useful, the same is true of systems developers. We are just as unlikely to persuade formal methods practitioners to change their way of working (so that it becomes more like that of UCD developers for example) as we are to change working practices of UI designers. Our second requirement is, therefore, that our integration techniques should work with existing formal methods rather than creating an entirely new formal process.

So, our intention is to integrate two existing modes of developing software by developing some bridge between the two processes. Ideally we want to do this in a way which does not add too much of a burden onto the already complex software development process, and in a way which can be adopted regardless of the actual formal methods being used within the development process (that is, it should not be tied to just one particular formal language).

Finally, our integration should encompass the entire process from initial design right through to implementation. As such we will need to include some notion of refinement in our work. We have seen in the research on plasticity and nomadic interface development that transformation techniques have been developed which allow consideration of UI designs at different levels of abstraction through to concrete implementations. What we wish to do is extend this idea and consider refinement of system and UI pairs together based on an understanding of what refinement for UIs means. This will require us to find ways of joining together system and UI descriptions (which will in turn mean describing them in some common language).

In summary then, although much of the existing research provides partial solutions to what we are trying to achieve, none of them fully satisfies the requirements we have outlined above. It is not unreasonable to expect that some of the work described be used alongside the methods we will propose, particularly in the areas of formalising task analysis prior to designing UIs. However, we consider this in chapter 8 as part of future work rather than in the main body of this thesis.

In the remainder of this chapter we discuss user-centred design and formal methods in relation to how they will be used within our research. We conclude with some background information on software development in the real world and a summary of the context of our research.

2.2 User-Centred Design

User-centred design is a term used to describe a collection of design techniques which ensure that when we develop software (and in particular the user interface to that software) the needs of the user remain central to the process. The use, and acceptance, of UCD as a practical and appropriate design methodology continues to increase, and has led to the evolution of the usability professional and any number of usability companies, websites, and books

devoted to its promotion and use. In a 2005 study of software developers to discover attitudes and uses of UCD in practice, Mao *et al.* [102], found that in the opinions of their respondents UCD has a significant impact and continues to gain acceptance (driven partly by growth in areas such as e-commerce).

As the basis for describing what we mean by user-centred design we refer to the ISO standard for human-centred design processes for interactive systems [1]. This standard describes user-centred design as a multi-disciplinary activity, which incorporates human factors, and ergonomics knowledge and techniques with the objective of enhancing effectiveness and productivity, improving human working conditions, and counteracting the possible adverse effects of use on human health, safety and performance.

The standard gives descriptions of four core activities for UCD, these are:

- understand and specify the context of use
- specify the user and organisational requirements
- produce design solutions
- evaluate designs against requirements

We therefore refer to UCD techniques as methods which satisfy any of the four core activities and which contribute to the development of effective user interfaces (where effectiveness is determined by meeting the criteria outlined.)

There are different stages in the UI design process, which is reflected in the four core activities. To begin we must gain an understanding of how and where the software will be used. The ways in which we can gather this domain knowledge may include activities such as performing ethnographic studies, running focus groups with users, interviewing users and brainstorming sessions. Once these activities are completed the designers should have a clear understanding of how the software will fit into the existing work practices of the users and what their expectations of the system are.

The next step is to identify the requirements and begin to consider the implications of the types of tasks users wish to perform. We have already

discussed task analysis (and in particular the formal methods which exist for this) which is one of the activities we expect will be undertaken at this stage. In addition we may use the results of our context-gathering work to inform these tasks with a wider understanding of what else the software is required to do over and above the identified user tasks (for example, it may be required to interact with existing software applications within an organisation).

Throughout our research when we talk about User Centred Design we mean that we expect that these two stages (and the others we will discuss next) have been followed. In terms of our integration requirements between UCD and a formal development process we do not make any links until we have reached a stage where these first two activities have been completed and a full understanding of the users' needs and their requirements has been developed. This corresponds to the gathering of the functional requirements in the formal process which we discuss in section 2.3.

Now the design moves into the third activity, that of producing design solutions. This is an iterative design process which encompasses prototyping in all of its forms. It is at this stage, when we begin to realise actual designs for the UI, that we wish to begin our integration. Producing design solutions results in physical artefacts representing the design (prototypes) which we can use as the basis for the integration (although we may not know how to do this yet). The importance of prototyping is well known amongst the UCD community. It is a fast, easy way to communicate design ideas to users, whilst at the same time providing an (often incremental) basis for the final UI design. The purpose of prototyping is not to produce fully-working design ideas where every aspect of the software is considered, but rather prototypes are an abstraction by way of describing the visual appearance of the system with an understanding of how this supports user tasks. They do not, therefore, include consideration of underlying functionality of the system, and it is at this point therefore that we need to begin to ensure that there is a compatibility between these designs and between the formal process we describe in section 2.3.

The final UCD activity of evaluating designs against requirements actually takes place throughout the iterative design process. The use of prototyping ensures that users are able to provide feedback and consider whether or not the proposed solutions will satisfy their needs. Designs are then updated to reflect their feedback. Again, it is important to consider the implications such changes may have as they may affect the overall solution. Our link between prototyping and formal design, therefore, should encompass the iterative nature of this part of the design process and be able to take on board any changes made and ensure that this does not introduce problems or incompatibilities. User interface evaluation also takes place once a final implementation has been produced and is generally referred to as usability testing. If changes are required at this stage we must ensure that they are appropriate in terms of the specified system and that we have some way of maintaining the link back to the earlier designs in order to check this. That is, if we have somehow proved some properties of our earlier designs and found ways to guarantee they correctly reflect the design of some formal specification we need a way to make sure this is preserved.

2.3 Formal Methods and Formal Notations

Formal methods are used to support reasoning about models or specifications of software prior to its implementation. The languages and notations used to build the models and specifications are themselves formal, that is they have a well-defined syntax and semantics which makes their meaning unambiguous. In addition they have a logic which allows us to manipulate the language in order to perform proofs *etc.* The purpose of using formal methods is to be sure that the software we build will behave as expected under all circumstances, that is it will be both robust and correct.

We have already made the point in chapter 1 that there are many different languages and notations which can be described as formal. These can be used

in different ways, but their essential purpose is the same, that of satisfying ourselves that we have correctly modelled the required systems and that subsequently we implement them correctly. In terms of the integration of formal methods and user-centred design processes it is not our intention to try and make a link between one specific formalism and one particular set of UCD techniques, but rather to provide a more general framework which can be used irrespective of the formalisms chosen or the design approach. In saying that, however, we do need to choose a formalism to explain our research and for this purpose we will use both the language Z [2] and the language μCharts [79].

Our reasons for choosing Z and μCharts are many. Z was initially developed in the 1970's by the Programming Research Group at the Oxford University Computing Laboratory and is now defined by an international standard [2]. It has been used in large real-world projects, such as the specification of the Mondex electronic purse [90], and the development of a reusable software framework for oscilloscopes for Tektronix [29]. There are many different support tools available for Z, including type-checkers, theorem provers and graphical editors. In addition, the Community Z Tools project (CZT) [27] is working to provide a common basis for Z tools allowing easy integration. We have experience of using Z both in earlier work which is related to this research [13] and more generally. The theorem prover we use in our research is Z/EVES [83].

μCharts is a visual, Statechart-like language used for describing reactive systems. It has a formal semantics given in Z as well as a refinement theory [80]. We will explain our reasons for using this language in more detail in chapters 5 and 6.

When we refer to a formal method within our work what we mean then is the process of specification, verification, validation and refinement using Z. Just as with the UCD design, there are several defined steps within this formal process. We begin with requirements gathering which forms the basis of the initial specification. The requirements are in some natural language description

and so must then be "translated" into the required formal language to become a specification. Before we can begin to prove properties about this specification we need to make sure that it correctly describes the requirements and so we must perform some validation to ensure that this is indeed the case. Definitions for the terms 'verification' and 'validation' vary (Somerville, for example, gives several, subtly different definitions within the same text [88]). In general we will consider validation to be the process of ensuring that the specification meets the requirements and verification to be the process of ensuring that the implementation satisfies the specification.

The first step of validation is often a manual process where we check that the specification is complete, that is, each of the requirements is represented somewhere within the specification. We can then begin to prove properties about the specification to ensure that we have specified each of these requirements correctly relative to the requirements. For example, if we have a requirement that a particular function should perform a particular task we want to prove that we have specified the function so that this task will always be performed correctly.

Validation is an iterative process, it may take several passes before we are satisfied that what we have is a correct specification. Once we have reached the stage where we are satisfied that we have correctly specified the requirements we are at an appropriate point to integrate this with the UCD process.

We begin our integration, therefore, when we have a completed formal specification and some design prototypes for the user interface. Before we move on to the next stage of development, that of transforming our abstract descriptions in a concrete implementation, we want to check that both parts of the system description (specification and prototype) are consistent. This will form the first part of our research, finding some way of performing this step of the integration.

The final stage of development is that of implementing the software. In a formal process this might be done by refinement, which provides a way of

transforming the specification into an implementation which is then guaranteed to be correct. Another possible approach is to build the system (somehow) and then verify it, that is, make sure it satisfies the specification. We discuss the process of refinement next.

2.4 Refinement

The process of transforming an abstract, formal description of a system (*i.e.* a specification) into an implementation which is correct relative to that specification is called refinement. In chapter 1 we have given some of the background and history of refinement and outlined examples of work which is of particular interest to us in the context of this research. These works are either related to refinement as it pertains to the use of Z, ([30], [104] [46] and [47]), or to some underlying principles of refinement which we find useful when we come to consider the meaning of refinement for UIs [65].

We have described above the different stages of development for the UI and system and the point at which we wish to begin integrating the two. The next stage for both parts of the development is moving towards an implementation. Just as the previous parts of both design processes have been iterative, so we assume that the refinement stage will also be iterative, and that it will move incrementally towards the final solution (which we have previously described as stepwise refinement).

Within the language Z, for example, there are already defined techniques for performing this refinement, however we have no such predefined process for the UI. Assuming that we have been able to integrate the UI prototypes with the formal specification and ensured that they describe the same system we similarly need to ensure that the final implementation also satisfies this. That is, we need to find a way of both considering refinement from UI design to implemented interface and finding a way of integrating this UI refinement with existing system refinement techniques. If we are able to produce such a

refinement method then we will have met our goal of integrating our formal and informal (UCD) software development processes from the point of creating the initial designs through to the point of implementation.

2.5 Design in the Real World

We have stated that we are interested in integrating formal methods and UCD because they are software development techniques which we believe are important and which appear to have a natural correspondence in terms of their goals. However, we can also show their respective importance in software development in the real world and as such our research is not purely academic but has relevance within the software development industry. The importance of both formal methods and UCD individually within industry leads us to believe that the benefits of an integrated method are likewise an important consideration in industry.

It is not enough to have an intuition that these things are important in software development generally: we were interested in finding out actual practices and principles currently used within the software industry. As such, in 2006 we undertook a survey of software developers within New Zealand to try and determine what approaches were being taken to software development and whether there was any common theme among developers. We subsequently compared the results of this survey with previously undertaken, larger-scale surveys in order to consider the topic from a more global perspective.

Our own survey targeted two different groups of participants: the first was post-graduate students enrolled in a New Zealand university who were developing software as part of their research; the second group was professional software developers working in New Zealand. We are most interested here in the responses from the professional software developers as they give a more accurate reflection of current working practices in New Zealand (our intention in also targeting students was to try and gain some understanding of what

the emerging practices may be based on potential future software developers' attitudes).

Unsurprisingly, one of the things we discovered is that there is a wide variety of different methods and approaches taken to developing software (irrespective of company size or software domain). Within these variances, however, the common themes of UCD and formal methods usage can be seen to play a significant part.

71% of the professional respondents used at least some formal methods within their development process (where 'formal method' here is defined as any method the respondent themself considers formal). Of the 29% who did not over half acknowledged that it was something they felt they should do. Over half of all development time was spent on the user interface to the system with over 80% of respondents using one or more UCD techniques. While the sample size for the survey was small (24 professional developer respondents) and limited to New Zealand, we subsequently examined our results in light of larger, world-wide surveys which have been undertaken and found that our results were not dissimilar. One particular point of interest in Cusumano *et al.'s* study [26] was that there are vast regional differences in usages of formal specification and formal design documents, with countries such as India having a much higher use than, for example, the USA. A survey of UCD practitioners by Vredenburg *et al.* in 2002 [102] found that task analysis and prototyping were the most popular methods used and that lo-fidelity methods were perceived more cost-effective. This is reflected in our results where paper-prototyping was the most commonly used UCD method.

Neither our own survey nor any of the others we examined undertook to find out if there was any attempt at integration between use of formal methods and UCD, however we rely on the literature presented in section 2.1 to outline what approaches may be taken in this respect and explain how this differs from the approach of our research. We present a summary of the results of our survey in Appendix A.

2.6 Conclusion

In this chapter we have given an overview of the existing research in the area of formal methods and user interface design and shown how this work differs from our own in both intention and approach. Much of the work described attempts to either formalise the UI design process by incorporating interactive concerns and user requirements into a formal specification, or attempts to bring formality into parts of a UCD process. While there are undoubtedly benefits from both such approaches, they do not satisfy our requirements, which are to integrate formal methods and UCD in a manner which allows us to retain the existing benefits both offer. As such, we do not wish to try and change the way in which formal practitioners and UCD designers go about their jobs, but rather find ways of creating a link between the two approaches taken and the different artefacts they produce. In particular, we want to ensure that the visual design of the UI is driven by the user requirements (both functional and usability) rather than being the result of transforming some formal UI model.

We have explained what approach we will take to the use of formal methods and UCD in our research, in particular noting that although we do not wish to tie our methods to one particular formalism we will adopt the Z language for the purposes of explaining our work. We have given a context of a Z-based formal process and a UCD process and described how we wish to integrate the two at the point at which we have a completed formal specification and have begun to produce UI design prototypes. We will subsequently continue this integration through the iterative transformation from design to implementation by way of a common refinement process. In this way we will be able to produce complete software solutions which have both the robustness and correctness guarantees of formal methods as well as the usability considerations of UCD.

Although our work has a theoretical basis, it has value both theoretically and practically in real world development situations. Both formal methods and UCD are important considerations in the software development industry

and providing robust integration methods is therefore a valuable contribution. Throughout this thesis we will provide practical examples to demonstrate how the models and methods we derive can be used, and in the next chapter we will describe some example software applications which we will use as the basis for these examples.

Chapter 3

Introduction of Examples

3.1 Introduction

We will now introduce two examples which we use in this thesis in order to demonstrate and explain our methods. The first example is a deliberately simple application with limited functionality. This allows us to describe all of the requirements (both functional and user) and present a full formal specification which is small and easy to read and understand. We can therefore use this example to explain our methods and techniques at all stages of the design process from initial design through to implementation.

The second example we will introduce is a description of a software application which was used as the basis for a case study into the use of our methods. This is a larger, more realistic example and we will therefore describe only the requirements for the application and subsequently introduce more detail from the case study as it is necessary.

3.2 Shape Application

The first example we introduce is the Shape application. We will use several variations of the Shape application in this thesis, but to begin with we will describe the basic application which we will refer to as Shape. The functional

requirements for the Shape application are as follows:

> The system can display three different shapes, a circle, a square or a triangle. When the system first starts up it displays no shape. There are three operations, one for each of the shapes, which cause that shape to be displayed. Once a shape has been displayed using a particular operation it remains on display until a different operation is selected. Repeatedly selecting the same shape will have no effect as the shape will remain displayed from the previous choice. The only time the system may display no shape is when it first starts up. There is no persistence built into the application and once it is closed it will restart in the no-shape display mode.

Based on these functional requirements we provide the following Z specification for the Shape application.

First we describe a type which can take the value of the required shapes for the system:

$$SHAPE ::= Circle \mid Square \mid Triangle \mid NoShape$$

Next we describe the system state which has a single observation, that of the currently selected shape:

System

$$selectedShape : SHAPE$$

In keeping with the requirements we create an initialisation operation which ensures the system starts in a state where no shape is selected:

SystemInit

$\Delta System$

$selectedShape' = NoShape$

Now we will describe three operations which allow the state to be changed so that each of the possible shape selections can occur:

SelectCircleOperation

$\Delta System$

$selectedShape' = Circle$

SelectSquareOperation

$\Delta System$

$selectedShape' = Square$

SelectTriangleOperation

$\Delta System$

$selectedShape' = Triangle$

This completes the initial specification of the Shape application. Because it is small we can satisfy ourselves by inspection that it correctly describes the system given by the requirements. It is also trivial to prove correctness and to prove it has certain properties, such as ensuring that once one of the select shape operations has occurred it is not possible for the system to be in a state where $selectedShape = NoShape$.

The user requirements for the shape application are very similar to the functional requirements, and are as follows:

> The user can choose to have one of three shapes displayed by the application: a circle, a square or a triangle. They should be able to select which of the shapes they want displayed and make this selection visible. Once they have selected which shape they want to be displayed it will appear on the screen and will remain displayed until the user makes another choice. Once the user has finished displaying shapes they can quit the application.

Based on the user requirements we can now identify the tasks that will be performed by the users:

- Choose to have a circle displayed
- Choose to have a square displayed
- Choose to have a triangle displayed
- Quit the application.

These tasks will form the basis of the UI design. We will introduce prototypes of designs for a UI to the Shape application in section 4.3.2.

3.3 PIMed

The PIMed tool was the basis for a case study performed in order to practically examine our methods on a piece of realistic software [17]. The purpose of the proposed tool is to support designers by providing an editor for presentation models and presentation interaction models (models we will present in chapter 4), it is therefore self-referential in that it is a tool to support the methods we are developing. The reason for using this example is that the software is something we consider important in the future of our work (support tools

are at least one way of encouraging developers to use new methods) and as such it provided a convenient and non-trivial example for us to develop (up to the point of implementation) using our methods, in order to determine how effective they are.

We will describe here the functional and user requirements, and in chapter 4 discuss some of the findings of the case study along with more detail of the work undertaken.

The functional requirements for PIMed are:

> PIMed is an editor which supports designers in creating presentation models and presentation interaction models (PIMs). It will be used to create, view, edit and print both types of model. Data will be preserved between uses of the application so that we can incrementally build up a collection of these models. Presentation models and PIMs can be created independently of each other but we can also use presentation models already in the system as the basis for developing a new PIM with the pair then remaining linked. Information will be stored in a way which reflects the hierarchical and component-based nature of the models. It will, therefore, allow for declarations to be entered and stored independently from completed models and used as required. Similarly the described presentation models should exist both as detailed, complete models in their own right and as components within conjunctions enabling them to be built up into larger models. The tool should in all respects follow the defined syntax and semantics for the models. The editor will have a graphical nature allowing PIMs to be viewed and edited graphically.

These initial requirements were used to form the basis of a functional specification. It is not useful to present the specification in its entirety here, but we will describe any relevant parts as necessary throughout this thesis.

The user requirements for PIMed were derived from discussions about how such a tool could be used within the design process and what sort of functionality would be most useful. This led to the development of the following key tasks which the tool should provide to users:

- Create a new presentation model
- View an existing presentation model
- Print a presentation model
- Edit a presentation model
- Create a new PIM from an existing presentation model
- Create a new PIM from scratch
- View an existing PIM
- Print an existing PIM
- Edit an existing PIM

Each of the key tasks identified above were subsequently broken down into sub-tasks to create a task hierarchy. For example, the requirement to create a new presentation model is broken down into the following task hierarchy:

- Create a new presentation model
  - Add declarations
    - Add model names
    - Add widget names
    - Add behaviour names
  - Add widget
    - Select widget name
    - Set category
    - Select behaviours
  - Compose presentation models

    * Select presentation models
    * Link models

The task hierarchy was used as the basis for the UI designs. Designs for 27 different windows and dialogues were prototyped, we will provide examples of these at appropriate points within this thesis.

Chapter 4

Models

4.1 Introduction

In this chapter we introduce the models which were developed as the first step of integrating formal methods and UCD. As we have previously stated, this first stage of integration is concerned with informal design artefacts, such as UI prototypes, and formal artefacts, such as specifications, and our aim is to find some way to formally link the two.

In chapter 2 we described a number of different UI models and highlighted some benefits and disadvantages of their use. There is a difference between these types of models and those we will describe in this chapter. Rather than creating a formal model of a UI and then using that as the basis for subsequent designs, we are taking existing designs which have been generated following a UCD process and trying to find a way to describe those designs formally. This will allow us to gain the benefits of having UI models (such as being able to prove properties about the UI) but without the disadvantages, such as having to persuade UI designers to change their working practices.

An important consideration in the development of such models is that they should not be overcomplicated and should not increase the burden on developers by requiring them to understand a difficult notation or model development process. The aim is to develop a model that is just formal enough, by which

we mean it meets all of the requirements we expect of any formalism (in terms of syntax, semantics and logic) and can therefore be used within a rigorous process, as well as including as much detail as necessary to make the link between formal and informal processes, whilst remaining simple enough to be easily understood, and used, by designers.

4.2 How to Formalise the Informal

Our goal is to create a formal model of an informal design artefact. Specifically we need to capture the information conveyed by UI designs in a way which enables them to be linked to a formal system specification. UI prototypes are used by designers both to suggest how parts of the UI will appear and to give users some indication of how they can interact with a system. Prototypes may be paper-based (usually in the form of sketches) or they may be produced using computerised drawing packages or interactive development environments (IDEs) (where some level of functionality may also be included) or even by specialist prototyping tools such as SILK [56]. Irrespective of how they are instantiated the intention behind all prototypes is the same: to produce an initial design of appearance and interaction possibilities which can be used to communicate ideas.

Prototypes generally start as a design sketched on paper (or created with a prototyping tool) and can initially be used to explain to users how they can interact with the system via the UI and what the behaviour will be when they interact. As such, the visual picture given by the prototype tells only part of the story. The design shows how the UI may look but is not enough on its own to convey all of the information it contains; by itself its meaning may be ambiguous. We require some additional information and context in order to be clear about what is being described. When we talk about the meaning of a prototype we are talking about what the implemented UI is supposed to do, *i.e.* what its behaviour is. When a designer shows a prototype to a user they

describe the visual elements and what their purpose is. We consider this to be the *narrative* that goes with a prototype. The narrative is the explanation the designer gives to the users as to how they can interact, what each part of the UI does and what the behaviour is that the system will exhibit when such interaction occurs.

Subsequently the prototype may evolve into something which a user can interact with, with the assistance of a designer who mimics the effects of the user's interactions by moving elements around or introducing new parts of the UI as required. So each part of the UI is prototyped (this can still be achieved using paper) and a person acting as the computer enables a user to explore the interface via interaction. Snyder [87] provides a detailed description of all elements of prototyping and the various types of activities which they can be used for in order to enable users to better understand the designs. Regardless of how a user interacts with a prototype (via discussions, or simulated interaction) there is still information required about the behaviour of the interface being designed which is not captured by the prototype itself.

In order to consider formally what a prototype means, we therefore need both parts of the information, the visual design (which we already have in the prototype) and the narrative. If we can formalise the narrative in some way we can use it to understand the behaviour of the UI which will then enable us to check this against our proposed system specification. What we want to find out is, does the proposed UI suggest the same behaviour we have described formally in our specification? There are obvious problems if this is not the case. It may be that there is behaviour suggested by the UI which is not part of the underlying specification, in which case it will never be implemented and there will be a mismatch when we come to join together the implemented functional core with the UI (imagine pressing a button on a UI which does nothing because its intended behaviour was never implemented).

4.3 Presentation Model

We introduce a model which formally describes a prototype of a UI design, which we call the *Presentation Model*. The presentation model is designed to capture the "what" and the "how" of UI design: *what* is the behaviour of the UI and *how* does a user access that behaviour? There is, of course, another consideration, the detail of appearance and layout, but this is often explicitly captured by the prototype itself (the purpose of prototypes within a UCD process is often to convey at least some idea of these details) and so we do not need to include it in the model. The presentation model describes the design artefact in terms of the interactive components of the design (which we will refer to as *widgets*), the nature of these widgets, and the behaviour they invoke.

Note that widgets do not need to be visually materialised on the UI. It may be that the software in question accepts voice commands, in which case we define widgets which generate/respond to behaviours just as we do if they are physical widgets. Similarly, there may be options tied to interaction devices (such as allowing a user to right-click using the mouse buttons anywhere on the screen) which causes certain behaviours, in this case we describe the whole area as a widget which has the respective behaviour. Likewise there is no assumption that the widgets we describe must be navigated to by a cursor via a mouse/keyboard and then activated by clicking (so we are not tied to the WIMP model[1]), just that there is some way for the user to invoke the associated action. For example, devices such as the Nintendo Wii and Apple's iPhone rely on motion detection for interaction inputs, but these can be included in the hierarchy by considering their effects in the same way as more traditional widgets. This is important in the ever-changing world of interfaces and input modes and allows our methods to be used irrespective of the interaction model.

[1] interaction based on windows, icons, menus and pointing devices

Figure 4.1: Motion Detection for iBeer on iPhone

The presentation model categorises widgets using the widget categorisation hierarchy given in [13] as its basis. Widgets are grouped into one of three categories based on their high-level behaviour, for example do they cause actions to occur when a user interacts with them, such as a button (which is in the Event Generator category), do they provide information back to the user, such as a message label (Event Responder) or do they act as a grouping mechanism for other widgets, such as a menu (Container). Each of these categories is then further broken down into sub-categories in a tree structure until eventually at each leaf node we can place an actual widget. Any widget can be positioned in the hierarchy by determining its general behaviour and following the tree hierarchy until a suitable node is reached. The three hierarchies are given in Appendix B.

4.3.1 Syntax and Semantics

We start by giving a description of the syntax of presentation models and an explanation of the different parts of the model. Presentation models consist of

declarations and definitions.

$$\langle pmodel \rangle ::= \langle declaration \rangle \langle definition \rangle$$

The declarations introduce the four sets of identifiers which can be used within the definitions.

$$\begin{aligned} \langle declaration \rangle ::= \quad & PModel\{\langle ident \rangle\}^{+2} \\ & WidgetName\{\langle ident \rangle\}^{+} \\ & Category\{\langle ident \rangle\}^{+} \\ & Behaviour\{\langle ident \rangle\}^{*3} \end{aligned}$$

A definition consists of one or more identifiers for presentation models and expressions which give the values for those identifiers.

$$\langle definition \rangle ::= \{\langle pname \rangle is \langle pexpr \rangle\}^{+}$$
$$\langle pname \rangle ::= \langle ident \rangle$$

Each expression is either a widget description, a presentation model identifier concatenated with another expression, or a presentation model identifier. This allows for a modular description of UI designs where each component (by which we mean individual window or dialogue) is described by a presentation model and the overall UI for the entire system is described by concatenating the component models.

$$\langle pexpr \rangle ::= \{\langle widgetdescr \rangle\}^{+} \mid \langle pname \rangle : \langle pexpr \rangle \mid \langle pname \rangle$$

A widget description consists of a triple, the widget name, the category and the set of behaviours associated with the widget. Each of the identifiers for these comes from the definition sets.

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

$\langle widgetdescr \rangle ::= (\langle widgetname \rangle,\ \langle category \rangle\ , (\{\langle behaviour \rangle\}^*))$

$\langle widgetname \rangle ::= \langle ident \rangle$

$\langle category \rangle ::= \langle ident \rangle$

$\langle behaviour \rangle ::= \langle ident \rangle$

An example of a legal presentation model is then:

PModel	*MainApp WinA WinB*
Widgetname	*ControlOne SelTwo ControlThree*
Category	*ActCtrl SValSelector*
Behaviour	*DoActionA DoActionB DoActionC*
WinA is	$(ControlOne, ActCtrl, (DoActionA))$
	$(SelTwo, SValSelector, (DoActionB))$
WinB is	$(ControlThree, ActCtrl, (DoActionC))$
MainApp is	*WinA : WinB*

This model describes a UI design such as that given in figure 4.2.

Each component presentation model consists of a set of widget tuples such that the : operator acts as the set union operator. The expression:

MainApp is WinA : WinB

means, therefore, that *MainApp* consists of the union of *WinA*'s widgets and *WinB*'s widgets. In the case where two conjoined models contain a common element we will get the expected result that the union of these models will contain a single instance of that element. This may initially appear problematic as it suggests we 'lose' one of the widget descriptions. However, recall that the model is intended to capture the behaviours embodied in the prototype along with associated widgets describing how the behaviour is accessed and we have

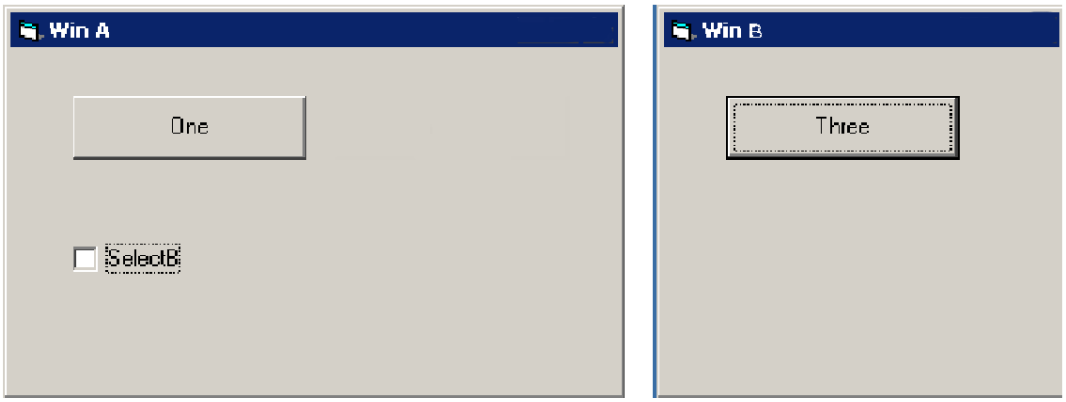


Figure 4.2: Example UI Design

not lost any of this information in our result. We will consider this in more detail shortly.

We now give the semantics of the presentation model. Firstly, we describe the complete model of a design as an environment, ENV. This environment is a mapping from the name (from the Ide of identifiers) of some presentation model and its parts to their respective values:

$$ENV = Ide \rightarrow Value$$
$$Value = Const + \mathbb{P}(Const \times Const \times \mathbb{P}\, Const)$$
$$Const = \{\overline{v} \mid v \textit{ is an identifier}\}$$

We use the bar over an identifier name (as in $\overline{ide}$) to indicate the actual entities we are describing. So we consider there to be a difference between the abstract entities described and the notation used to describe them. $\overline{ide}$ gives us a mapping from a name (which is part of the notation) to the entity it refers to. This is a uniform way to describe entities within denotational semantics (see [42] for example) where we do not commit to any particular instantiation but treat these abstract things as if they are real such that the identifiers act as placeholders.

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

$$[\![\_]\!] : \langle pmodel \rangle \rightarrow ENV$$
$$Dc : \langle declaration \rangle \rightarrow ENV$$
$$Df : \langle definition \rangle \rightarrow ENV \rightarrow ENV$$
$$Expr : \langle pexpr \rangle \rightarrow ENV \rightarrow Value$$

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

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

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

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

where ρ represents the current environment.

$$Expr[\![E\ Es]\!]\rho = Expr[\![E]\!]\rho \cup Expr[\![Es]\!]\rho$$
$$Expr[\![\psi : \phi]\!]\rho = Expr[\![\psi]\!]\rho \cup Expr[\![\phi]\!]\rho$$
$$Expr[\![(N\ C\ (b_1 \ldots b_n))]\!]\rho = \{(\rho(N)\ \rho(C)\ \{\rho(b_1) \ldots \rho(b_n)\})\}$$
$$Expr[\![I]\!]\rho = \rho(I)$$

We can also define semantic functions which allow us to extract information from the model, for example we define the following which allows us to obtain

the set of all behaviours of a presentation model:

$$B[P]$$

where

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

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

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

In Appendix C we give an example of using the denotational semantics to instantiate the example presentation model given above for the design of figure 4.2.

4.3.2 Properties and Uses

The presentation model describes widgets using a triple consisting of the widget name, its category and its behaviours. The assigning of names to widgets is not a random process (where any name is acceptable) but rather is based on one of two things. Either the widget in question is visually represented on the UI with some label indicating its purpose, in which case the associated label is used as the name, or a name which gives some indication of underlying behaviour

is used. So, for example if we have a widget of the UI which is a button labelled "Quit" then we expect that the name given to that widget in the presentation model is "Quit" (or "QuitButton", "QuitCntrl" *etc.*). Similarly if we have a widget which is labelled with an icon (such as a printer icon on a widget which enables printing behaviour) then we expect the name given to that widget would be "Print" (or something similar). This ensures that there is an easily identifiable mapping between the widget descriptions of the presentation model and the widgets of the prototype which makes it easier for both UI designers and system developers to relate the model to the prototype.

Although we describe behaviours of each widget within a single behaviour set in the widget triple, we can in fact expand the description by considering different categories of behaviours, and we will later rely on this for a more complete understanding of UI behaviour. There are two types of behaviour of a widget. The first is what we will call an *Interaction Behaviour* (or *I_Behaviour*). *I_Behaviours* are those behaviours which affect the UI itself, for example by changing some aspect of the visual appearance (such as resizing a window) or by opening a new window or dialogue. As such, *I_Behaviours* enable users to control their interactive experience by allowing navigation through the UI and control of the UI appearance.

The second type of behaviour is what we call a *System Behaviour* (or *S_Behaviour*). An *S_Behaviour* is any behaviour which affects the underlying system. These are the behaviours which allow a user to interact with the system either by accessing, changing, or obtaining information. Figure 4.3 shows the interactivity between user, UI and system in terms of the described behaviours.

Rather than adding an explicit categorisation for behaviours to the presentation model we instead rely on a naming convention such that interaction behaviours are given names prefixed with I_ and system behaviours have names prefixed with S_.

To explain some of the properties of presentation models we now introduce

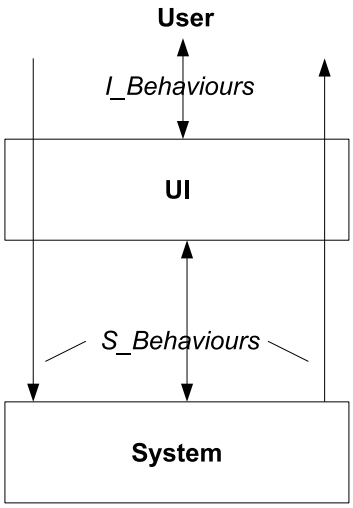


Figure 4.3: Interaction via Behaviours

an example based on the Shape Application described in chapter 3. In figure 4.4 we give a preliminary design prototype for the UI of the Shape application. The presentation model for this design is:

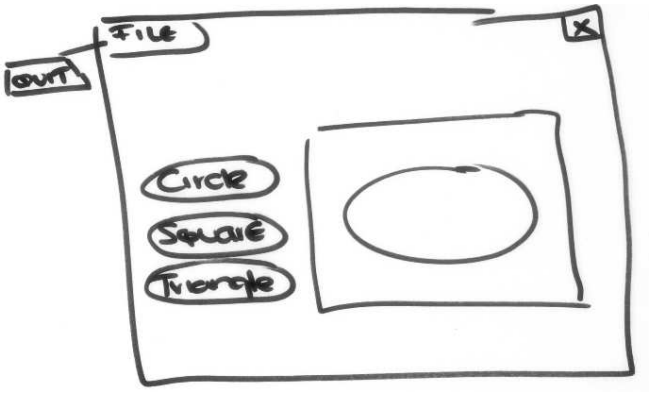


Figure 4.4: Prototype for Shape UI

PModel	*ShapeApp*
Widgetname	*Circle*, *Square*, *Triangle*, *SFrame*, *File*, *Quit*, *CBox*

Category	*ActCtrl*, *Container*
Behaviour	*S_ShowCircle*, *S_ShowSquare*, *S_ShowTriangle*, *Quit*

ShapeApp is	$(File, Container, ())$
	$(Quit, ActCtrl, (Quit))$
	$(Circle, ActCtrl, (S\_ShowCircle))$
	$(Square, ActCtrl, (S\_ShowSquare))$
	$(Triangle, ActCtrl, (S\_ShowTriangle))$

Notice that one of the behaviours, *Quit*, which is associated with two widgets, is not labelled as either an *I_Behaviour* or an *S_Behaviour*. We consider *Quit* to be a special behaviour as it terminates both the system and the UI. We expect that under normal circumstances all of our designs will include at least one widget with a *Quit* behaviour (unless the requirements are specifically that the user should not be able to terminate the application) and it is the only behaviour we will describe which is not designated as an I or S behaviour.

The Shape application presentation model has three *S_Behaviours*, which represent functionality which satisfy each of the user requirements. It is easy to see from this small example how the prototype design has been developed from the user requirements, as each of the behaviours has a direct mapping to the tasks outlined in the description given in chapter 3. The *S_Behaviours* are what enables the user to access the functionality of the application. As such we would expect that these behaviours would also be present in the system specification, *i.e.* we expect the described functionality of the system to support these tasks. Our aim is to show consistency between system and UI, which means they support the same user tasks and functionality either by way of UI behaviours or specified system operations. In order to determine whether or not the UI design and the system specification are consistent in their intended behaviour we will, therefore, create a relation between the *S_Behaviours* of the presentation model and the operations of the system specification.

The specification of the Shape application given in section 3.2 described

three operations, *SelectCircleOperation, SelectSquareOperation* and *SelectTriangleOperation* we now create a relation, which we call the presentation model relation (or *PMR*), which shows how the *S_Behaviours* of the UI design relate to these specified operations.

$S\_ShowCircle \leftrightarrow SelectCircleOperation$

$S\_ShowSquare \leftrightarrow SelectSquareOperation$

$S\_ShowTriangle \leftrightarrow SelectTriangleOperation$

ShapeApplicationPMR

The *PMR* is a total, many-to-one relation. In order to be sure that the UI behaviours define functionality which will exist in the system each *S_Behaviour* must be related to a specified operation. Ideally we would like the relation to be injective, that is each *S_Behaviour* be related to a distinct operation. However, in some cases multiple *S_Behaviours* may relate to the same specified operation which indicates different UI behaviours which perform the same task. We will discuss the implications of this in section 4.3.5. Conversely, we do not expect that every specified operation will be related to an *S_Behaviour*, as there will be operations which do not relate to functionality which should be available for the user, for example security aspects of a system.

It may be the case that the specification is described at a lower level of abstraction than the UI design so that a single behaviour of the UI is actually represented by a collection of operations in the underlying system, in such cases we rely on the use of either schema conjunction or composition within the specification (in our Z specification example) to build a higher level description. This allows us to describe a single operation consisting of these lower-level operations such that the relation can be built as described.

Within the widget description tuple we have a set of behaviours, *i.e.* it is possible for a widget to have more than one behaviour associated with it.

The meaning of having multiple behaviours depends on the category hierarchy the widget belongs to. For event generating widgets that have more than one behaviour the meaning is that all of these behaviours are invoked when the widget is activated (it does not indicate a choice between these behaviours). For event responding widgets that have more than one behaviour, the meaning is that they are capable of responding to any of the given behaviours in the set and will respond to whichever is present (determined either by user interaction of system activity), again it does not indicate choice. The ability of event responding widgets to respond to a number of different behaviours may suggest a level of nondeterminism within the UI design. However, this is not the case and will explain the role of nondeterminism in UI designs and presentation models next.

4.3.3 Nondeterminism in Presentation Models

Presentation models do not contain nondeterminism. At first this might seem an unusual property given that nondeterminism is a common method of providing abstraction in early specifications. Nondeterminism allows us to ignore non-essential details and postpone specifics until later in the development process. As such it is not unusual for our formal specifications to be nondeterministic. The nondeterminism is then resolved during the refinement process.

The presentation model describes an abstract view of a user interface, namely a prototype, and so we might consider that it is reasonable to expect nondeterminism to be used in the same way as in the formal model. In order to explain why this is not the case we need to consider the purpose of UI designs in a little more detail.

UI designs provide three types of information. They show how a user can interact with a system to access the system's functionality (via widgets which provide $S\_Behaviours$), they show how a user can interact with the UI itself (via widgets which provide $I\_Behaviours$), and they also give an indication of what the final UI will look like and what types of widgets will be used. We

use presentation models to describe the first two parts of this information as well as capturing widget-type information from the third.

Consider then how nondeterminism might be used in UI designs. In terms of the appearance of the UI it is quite likely that there will be nondeterminism present. This is because early designs are unlikely to fully describe every aspect of the UI's appearance (they abstract away details about background colours, font choice and size, precise layout *etc.*) We can consider this to be descriptive nondeterminism, that is, the designs are clear about the behaviour of the UI, but abstract about appearance.

Although we state that the designs are clear about the behaviour, it is again not unreasonable to consider that a prototype may not show a *complete* design. There may only be designs for some parts of the UI and so we expect that there would be some nondeterminism present in the behavioural description also. Such nondeterminism, however, is not included in the presentation model. To understand why this is so we can examine some examples of how nondeterminism may appear in presentation models.

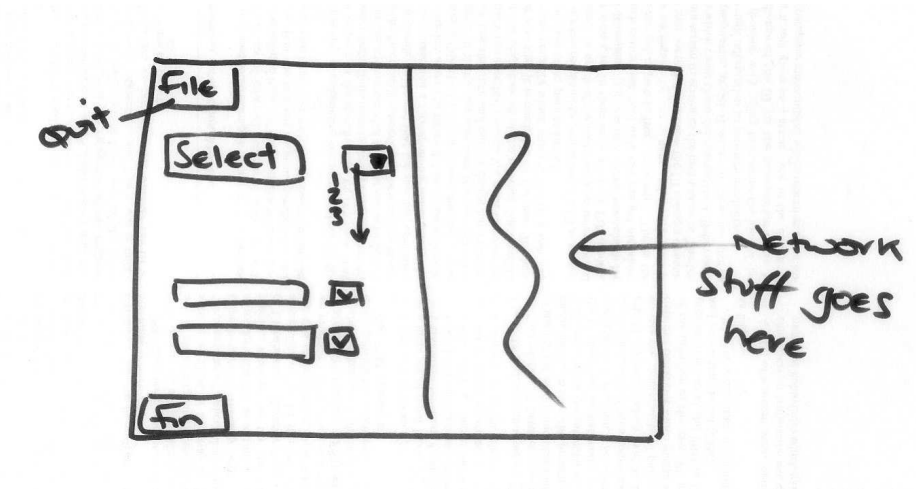


Figure 4.5: UI Design with Nondeterminism

Figure 4.5 shows the design of a UI which is only partially completed. The right-hand side of the screen is dedicated to interacting with a particular part

of the system and has not yet been considered (or is possibly being considered by some other designer). Whilst there are decisions about this design which have not yet been made and we could state that we do not care what happens in terms of the interaction with network behaviour widgets, this is not really a nondeterministic model. Rather it is a partial design where the UI is not yet complete.

PModel	*NonDet*1
Widgetname	*SelBtn FileMenu QuitItem NumList AField BField ASel BSel FinBut*
Category	*ActCtrl Containe rSValSel*
Behaviour	*S_SelectQuit S_SendVal S_SetA S_SetB S_Save*

*NonDet*1 *is*	$(SelBtn, ActCtrl, (S\_Select))$
	$(FileMenu, Container, ())$
	$QuitItem, ActCtrl, (Quit))$
	$NumList, SValSel, (S\_SendVal))$
	$AField, Container, ())$
	$BField, Container, ())$
	$ASel, ActCtrl, (S\_SetA))$
	$BSel, ActCtrl, (S\_SetB))$
	$FinBut, ActCtrl, (S\_Save))$

The presentation model contains all of the information for the described widgets given on the left-hand side of the design, but nothing for the undefined right-hand side. This is because the presentation model is intended to describe the narrative of the design: it provides a description of the intended interaction and behaviour for the design. In this case we have no information about the right-hand side of the design and there is, therefore, nothing to say about it in the presentation model. Functionality which is not full due to partial

definition is not nondeterminism. This is an example of how delaying design decisions does not lead to nondeterminism in the formal model of that design.

Figure 4.6 shows the design of a UI where all of the widgets have been defined, but behaviour for some of those widgets is not defined, *i.e.* the widgets themselves are nondeterministic. If we give the presentation model of this

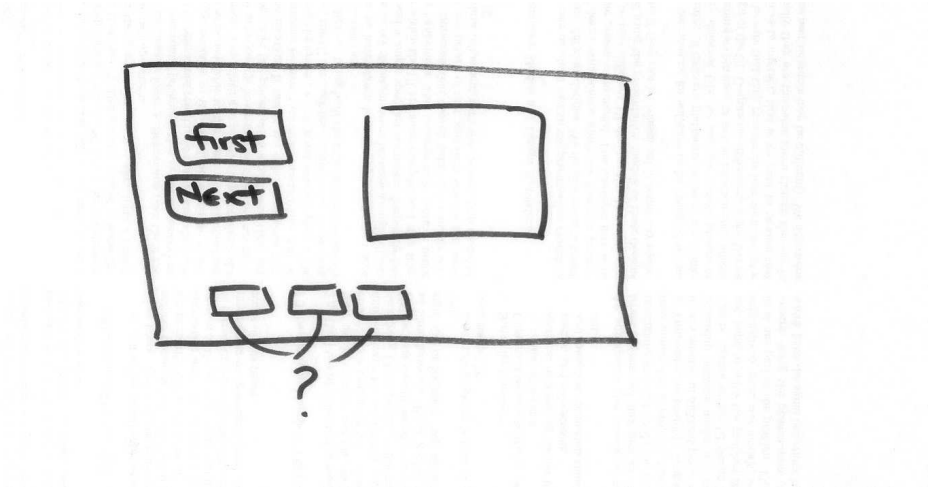


Figure 4.6: UI Design with Nondeterministic Widgets

design then the nondeterminism *does* carry over into the model because we have some widgets to describe, but do not yet know their behaviour.

PModel	*NonDet2*
Widgetname	*First Next Box BOne BTwo BThree*
Category	*ActCtrl SValResp*
Behaviour	*S_ShowFirst S_ShowNext S_Display*

NonDet2is	$(First, ActCtrl, (S\_ShowFirst))$
	$(Next, ActCtrl, (S\_ShowNext))$
	$(Box, SValResp, (S\_Display))$
	$(BOne, ?, ())$
	$(BTwo, ?, ())$

$$(BThree, ?, ())$$

The first problem with this presentation model is that we do not know the categories for $BOne, BTwo$ and $BThree$, making the model incomplete. We could give them a default category of *Widget* (being the top-most category from which all of the hierarchy trees derive), but we are still left with empty behaviour sets which may or may not be correct. However, such nondeterminism does not make sense in a design produced as part of a UCD process. The basis for the UI designs is the user-defined tasks which have been identified earlier in the design process. The prototypes and designs are intended as a first step toward describing how these tasks may be performed in the UI, and so the requirements are used to determine what needs to be provided via the UI and as the basis for the design.

In the example of figure 4.6 there are three widgets whose purpose is unclear, *i.e.* their inclusion is not driven by the user requirements. While it is possible that in some sorts of design the designer may start with a particular set of widgets and layout they want to use and fit the required functionality into this later, this is not the case for user-centred design. It does not make sense within such a process to randomly add widgets with no thought as to their behaviour. If the user requirements are themselves unclear then we would expect that the designer would either seek clarification from the users prior to developing the prototype, or that they would omit the behaviour they are unclear about and use the partial prototype as a method of clarification as they discuss it with the users. For these reasons we do not include the notion of such nondeterministic widgets into our design methods.

For another example of nondeterminism of designs we return again to figure 4.5. Within the presentation model there is a widget called *NumList* whose behaviour is *S_SendVal*. It may be the case that when we build the *PMR* for this model, the operation in the specification that *S_SendVal* is related to is defined nondeterministically. We cannot, therefore, state clearly what will

happen when the user interacts with this widget and so can consider this UI to be nondeterministic. Our presentation model, however, does not capture this nondeterminism; the behaviour of the UI is clearly defined and it is the underlying system specification which is nondeterministic.

As a final example in this section we present the design of figure 4.7. This dialogue window presents a list of names of members of a group and allows a user to see more details about any of the listed people by clicking on their name. There are two different types of group members, staff and students. These two group types have different recorded information and so depending on the type of the person listed different information is shown. The presentation model for this design is:

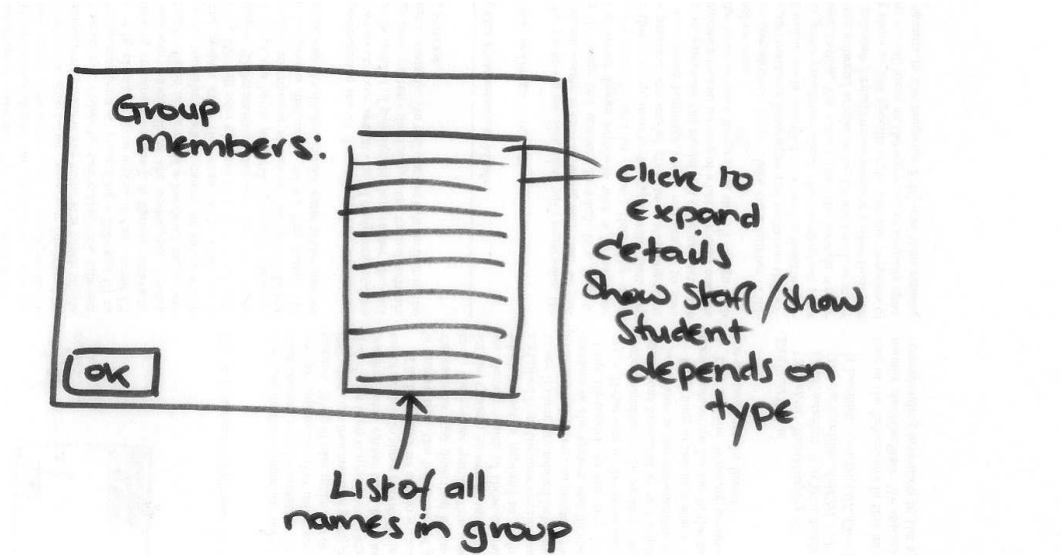


Figure 4.7: UI Design with Nondeterminism

PModel	*NonDet3*
Widgetname	*NameList OkButt*
Category	*ActCtrl SValSel*
Behaviour	*I_CloseDialogue S_ShowStaffDetail S_ShowStudDetail*

*NonDet*3*is* (*NameList*, *SValSel*, (*S_ShowStaffDetail*, *S_ShowStudDetail*))
(*OkButt*, *ActCtrl*, (*I_CloseDialogue*))

the *NameList* widget has two behaviours associated with it. We previously stated that when a widget which is in the responder category has multiple behaviours it responds to whichever of those behaviours occurs at any given time. For widgets in the generator category, however, we stated that multiple behaviours indicated that all of these behaviours would occur each time the widget is interacted with, but in this design it is not intended that all of the behaviours occur. Here, there is a choice of which of the behaviours occur when the widget is activated, and so the design is nondeterministic. If we consider this more carefully and think about how we will implement this design, it becomes clear it is not the case that the nondeterminism can be removed by choosing just one of the behaviours (*i.e.* it is not the case that it doesn't matter which of the behaviours occur), but rather that there must be some mechanism for selecting the correct one. This decision will not take place at the UI level at all, instead some underlying system operation must be used to check the type of person whose name is selected in order that the correct details can be shown. What we have, therefore, is an underspecified design which is in fact misleading. Rather than the behaviour of the widget being to display the relevant details, it is in fact to interact with the system to find out what the type is of the selected name. The system then performs the appropriate action in terms of information display. We make this clear in the design given in figure 4.8, whose presentation model is:

PModel	*NonDet*3*b*
Widgetname	*NameList OkButt*
Category	*ActCtrl SValSel*
Behaviour	*I_CloseDialogue S_CheckType*

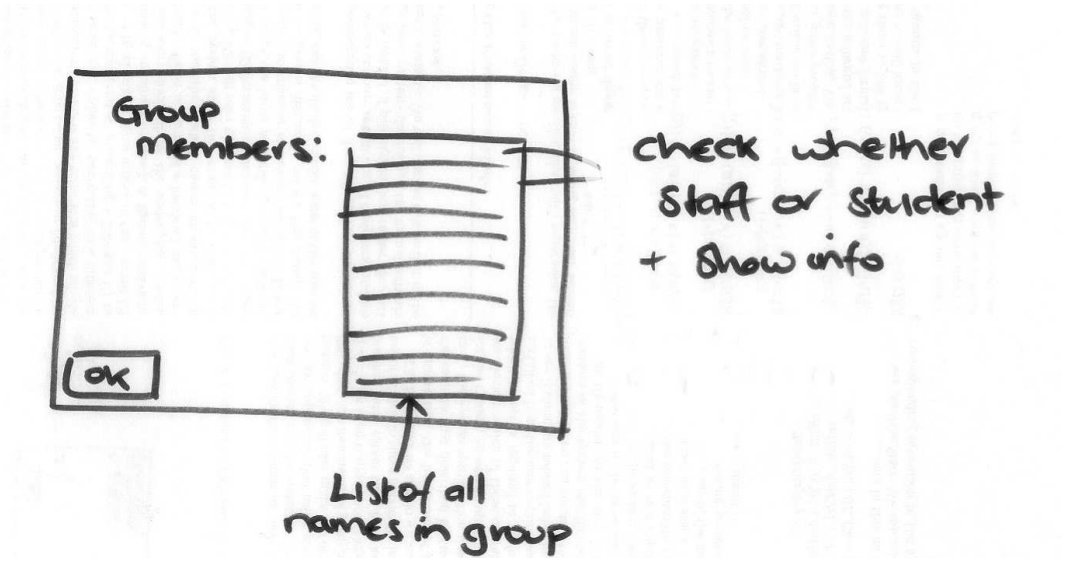


Figure 4.8: UI Design with Nondeterminism Removed

$$NonDet3b\ is \quad (NameList, SValSel, (S\_CheckType))$$
$$(OkButt, ActCtrl, (I\_CloseDialogue))$$

The system specification will describe an operation which checks the type of the person selected and displays information accordingly. This system operation will be related to the *S_CheckType* behaviour in the *PMR*. In this example, the presence of nondeterminism has shown that the design is not clear enough which leads us to consider the UI behaviour in more detail.

These examples show that descriptive nondeterminism in UI designs does not lead to nondeterminism in presentation models. Also, while behavioural nondeterminism may exist in UI designs, the presentation models of those designs do not include nondeterminism. When we use these models to consider notions of UI refinement we will not, therefore, consider removal of nondeterminism as one of the goals.

4.3.4 Presentation Models and Design Equivalence

Another use for presentation models is that of comparing different designs to see how alike they are. This enables us to take different UI designs for the same system, or different versions of the same system and determine whether or not they can be considered in some way equivalent.

Equivalence of UIs is based on the defined behaviours (as given in the presentation model) and as such we refer to it as *Behavioural Equivalence*. It is based on a comparison of the sets of behaviours of the presentation models of designs. Formally we state:

Definition 1 *If DOne and DTwo are UI designs, and PMOne, PMTwo are their corresponding presentation models, then:*

$$DOne \approx_{Beh} DTwo =_{df} B[PMOne] = B[PMTwo]$$

Behavioural equivalence allows us to determine if UI designs have the same behaviour, irrespective of their appearance or the widgets they use.

We can define two further types of equivalence based on the different types of behaviours. *S_Behaviour* equivalence ($\approx_{SBeh}$) where the sets of *S_Behaviours* are the same and *I_Behaviour* equivalence ($\approx_{IBeh}$) where the sets of *I_Behaviours* are the same. These are formally described as:

Definition 2 *If DOne and DTwo are UI designs, and PMOne, PMTwo are their corresponding presentation models, then:*

$$DOne \approx_{SBeh} DTwo =_{df} S\_B[PMOne] = S\_B[PMTwo]$$

Definition 3 *If DOne and DTwo are UI designs, and PMOne, PMTwo are their corresponding presentation models, then:*

$$DOne \approx_{IBeh} DTwo =_{df} I\_B[PMOne] = I\_B[PMTwo]$$

$S\_B[PMOne]$ and $I\_B[PMOne]$ are semantic functions in the same manner as $B[PMOne]$.

Behavioural equivalence is an important consideration in the determination of UI refinement. We will discuss this in detail in chapter 6.

4.3.5 Benefits of Presentation Models to UI Design

The initial purpose of the presentation model was to find a way of formalising informal design artefacts in order that they could be used in conjunction with a formal software development process. However the model also provides benefits to the UI design process over and above this integration and we discuss this next.

There are many different ways of defining what is meant by a *good* UI. These encompass such things as aesthetics, usability, learnability, robustness *etc.* Researchers interested in finding ways to develop better interfaces have attempted to categorise these properties in order to describe a core set of principles which should be adhered to in order to produce better UIs. Examples of such work include Shneiderman's "Eight Golden Rules of Interface Design" [86], Nielsen's "Top 10 Mistakes in Web Design" and other usability guidelines [98] and Norman's "The Psychology of Everyday Things" [66]. Within these works there is a core set of common properties which therefore appear to be important in this context. Some of these properties can be identified using presentation models and therefore the models provide a way to check for desirable UI properties at the prototype stage, rather than after implementation which is often when such checks are performed. There are then benefits due to creating the models over and above the intention of including UIs design in a formal process.

One desirable UI property which is described in the works mentioned above is that of consistency. Consistency allows users to learn how to use UIs more quickly (once they understand certain interactions and how the UI works consistency ensures those same interactions can be used throughout the UI) and also means that users do not encounter unexpected behaviour (where their experience of interacting with some parts of the UI leads them to expect a certain behaviour, but the UI then behaves differently). Shneiderman [86] describes this property as:

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

Shneiderman and Mahajan describe a collection of tools, SHERLOCK [62], which can be used to identify inconsistency (both textual and visual) in implemented UIs. The tools (which include such things as a dialogue box analyser and a button concordance tool) are adaptable and generalisable such that they are multi-platform and can be used (in theory) to examine any UI. The main disadvantage of this approach is that any errors or problems are discovered after the UI has been implemented which means it is often difficult and time-consuming to fix. Using our models, however, we can discover consistency problems prior to implementation when it is relatively straight-forward to make the necessary changes.

One way of providing consistency is by ensuring that terminology used throughout the UI is consistent. We can test for this within a UI design using the presentation model. The behaviours of the presentation model correspond to actions of the user (or the system). Where there are multiple ways to perform a particular action using different widgets, there will be repetition of a particular behaviour within the widget descriptions in the model.

We can identify common behaviours within the presentation model and examine the widgets which have this behaviour associated with them and use this to consider issues of consistency. For example, we may have a UI which has several different windows and each of these windows has behaviours in common. From the presentation models for each window we can identify those widgets which have these behaviours and check that there is consistency both in the way each window provides this functionality and in the naming conventions used.

Figure 4.9 gives an example of a design with multiple windows and shows partial prototypes for each of these. The common behaviours across each part

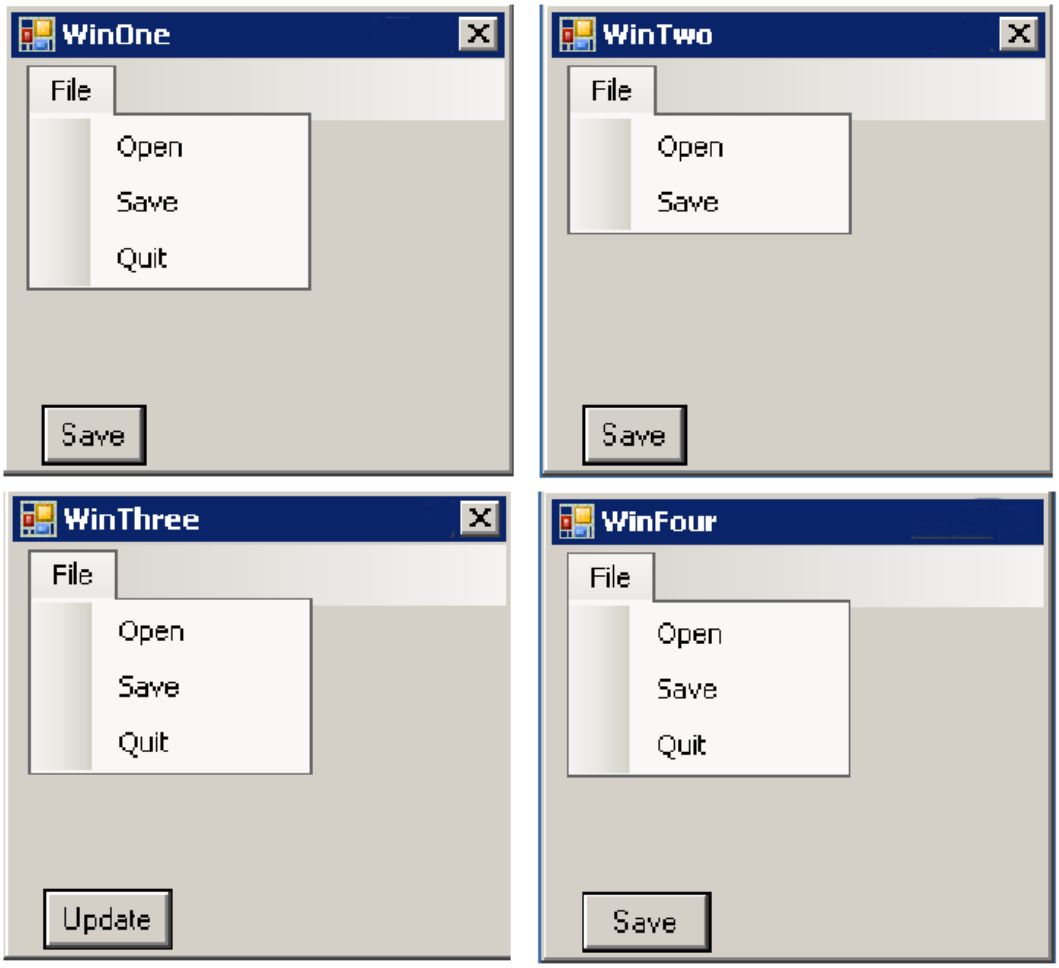


Figure 4.9: UI with Four Windows

of the UI are the ability to *Quit* the application, some functionality to *Open* a file, and *Save* functionality. The presentation models for each of these are as follows:

PModel	*WinOne WinTwo WinThree WinFour*
WidgetName	*FMenu Open Save Quit Update*
Category	*Container ActCtrl*
Behaviour	*S_Open S_SaveQuit*

WinOne is $(FMenu, Container, ())$,

$(Open, SValSel, (S\_Open)),$
$(Save, SValSel, (S\_Save)),$
$(Quit, SValSel, (Quit)),$
$(Save, ActCtrl, (S\_Save)),$
$(Quit, ActCtrl, (Quit))$

WinTwo is $(FMenu, Container, ()),$
$(Open, SValSel, (S\_Open)),$
$(Save, SValSel, (S\_Save)),$
$(Save, ActCtrl, (S\_Save)),$
$(Quit, ActCtrl, (Quit))$

WinThree is $(FMenu, Container, ()),$
$(Open, SValSel, (S\_Open)),$
$(Save, SValSel, (S\_Save)),$
$(Quit, SValSel, (Quit)),$
$(Update, ActCtrl, (S\_Save)),$
$(Quit, ActCtrl, (Quit))$

WinFour is $(FMenu, Container, ()),$
$(Open, SValSel, (S\_Open)),$
$(Save, SValSel, (S\_Save)),$
$(Quit, SValSel, (Quit)),$
$(Save, ActCtrl, (S\_Save))$

First we examine the widgets from each component presentation model which have the *S_Open* behaviour associated with them:

$WinOne : (Open, SValSel, (S\_Open))$
$WinTwo : (Open, SValSel, (S\_Open))$
$WinThree : (Open, SValSel, (S\_Open))$
$WinFour : (Open, SValSel, (S\_Open))$

Each window has one widget with this behaviour which has the same widget category and the same name. From this we can determine that the ability of a user to access this behaviour is consistent in each part of the UI. Next we examine the widgets with the *S_Save* behaviour associated with them:

$WinOne : (Save, SValSel, (S\_Save))$

$WinOne : (Save, ActCtrl, (S\_Save))$

$WinTwo : (Save, SValSel, (S\_Save))$

$WinTwo : (Save, ActCtrl, (S\_Save))$

$WinThree : (Save, SValSel, (S\_Save))$

$WinThree : (Update, ActCtrl, (S\_Save))$

$WinFour : (Save, SValSel, (S\_Save))$

$WinFour : (Save, ActCtrl, (S\_Save))$

Although each of the windows has the same number of widgets with this behaviour and the widgets have the same categories, in WinThree the name *Update* is used rather than *Save*. This suggests that this prototype contains some inconsistency which should be examined further (it may be the case, of course, that this is an intentional design choice but by identifying it from the presentation model we ensure that it is reconsidered to ensure this really is the case rather than an accidental oversight). Finally we examine the widgets with *Quit* behaviour:

$WinOne : (Quit, SValSel, (Quit))$

$WinOne : (Quit, ActCtrl, (Quit))$

$WinTwo : (Quit, ActCtrl, (Quit))$

$WinThree : (Quit, SValSel, (Quit))$

$WinThree : (Quit, ActCtrl, (Quit))$

$WinFour : (Quit, SValSel, (Quit))$

WinOne and WinThree both have two ways of accessing the *Quit* behaviour and are consistent with each other whereas WinTwo and WinFour both have a single, different way of accessing this behaviour. Again this indicates that further examination of this prototype is required in order to resolve this in-

consistency.

Of course, because the example we are using here is necessarily simple it may seem unnecessary to use the formal models to determine a problme which can be identified by examination of the four designs. However, in a non-trivial system the number of different windows and widgets make this a much more difficult task to perform by manual inspection of designs. Having to compare *every* widget of *every* part of a design in this manner is both error-prone and time-consuming, whereas the amount of work required when using the models does not increase significantly as the size of the UI increases.

Another UI property we can consider by using presentation models is that of reactivity. Shneiderman [86] describes this property as supporting internal locus of control:

> "Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders."

This is a subtle property which is closely linked to the layout of the UI, the behaviour of the overall system and the balance of the types of widgets used. Dix *et al.* [33] also describe this as avoiding pre-emptive UI behaviour which keeps the user in control rather than the system. Whilst the presentation model does not provide information about the layout-related considerations it can be used to analyse the categories of widgets which in turn provides information about how much of the UI provides user-driven behaviour and how much is responsive to the system. Widgets whose category is in the *EventGenerator* hierarchy are widgets which allow the user to interact and generate behaviour. If we examine the balance between the percentage of widgets which are *EventGenerators* and the percentage which are *EventResponders* this gives us some idea of how reactive the UI is to the user. For example a UI where 40% of widgets are active and 60 % are passive indicates that it is less user-driven than a UI where 70% of widgets are active. This allows us to compare reactivity of different

design options or different UIs and could be used as a measure of reactivity in general.

Apart from the benefits of using formal methods which we have described in chapters 1 and 2 there are other known benefits of creating formal models for systems, which are also true of presentation models. In order to create a formal specification it is necessary to have a thorough understanding of the problem to be solved, if this is not the case then the formal model will not be correct. This necessity means that designers must think through each part of the problem and understand its complexity at an early stage, making it more likely that the proposed solution will not only be correct, but will also be suitably designed in order to handle such complexity (rather than discovering difficulties at implementation time which require major changes to the design). Similarly, when we create the presentation model of a UI design we must think carefully about each widget of the UI and what the intention of the design is. Again, this process of carefully considering the design early on ensures we are clear about exactly what we are proposing with the design (otherwise we will struggle to build the model).

Another benefit of formal modelling is that it provides a different way of viewing a system which may expose different issues than the set of requirements or the final implementation do, where some aspects of the problem may be hidden. The presentation model provides the same benefit for UIs by also making clear things which are not necessarily obvious in the prototype alone. The case study introduced in chapter 3 provides several examples of these sorts of benefits.

Based on the user requirements for the PIMed application described in chapter 3, a total of twenty seven window/dialogue prototypes were developed. Each of these prototypes was subsequently described by a component presentation model, which can be conjoined to give the presentation model for the complete UI. During the development of the presentation models we found several things within the designs which required further clarification or

necessitated changes. In some cases this occurred where the *narrative* of the design was itself unclear, that is, on returning to the design the designer was unsure as to what their intention was with a particular widget. The prototype shown in figure 4.10 is the original prototype of one of the windows for PIMed. Three of the controls, the lists called PModel, Widgets and Behaviours, are

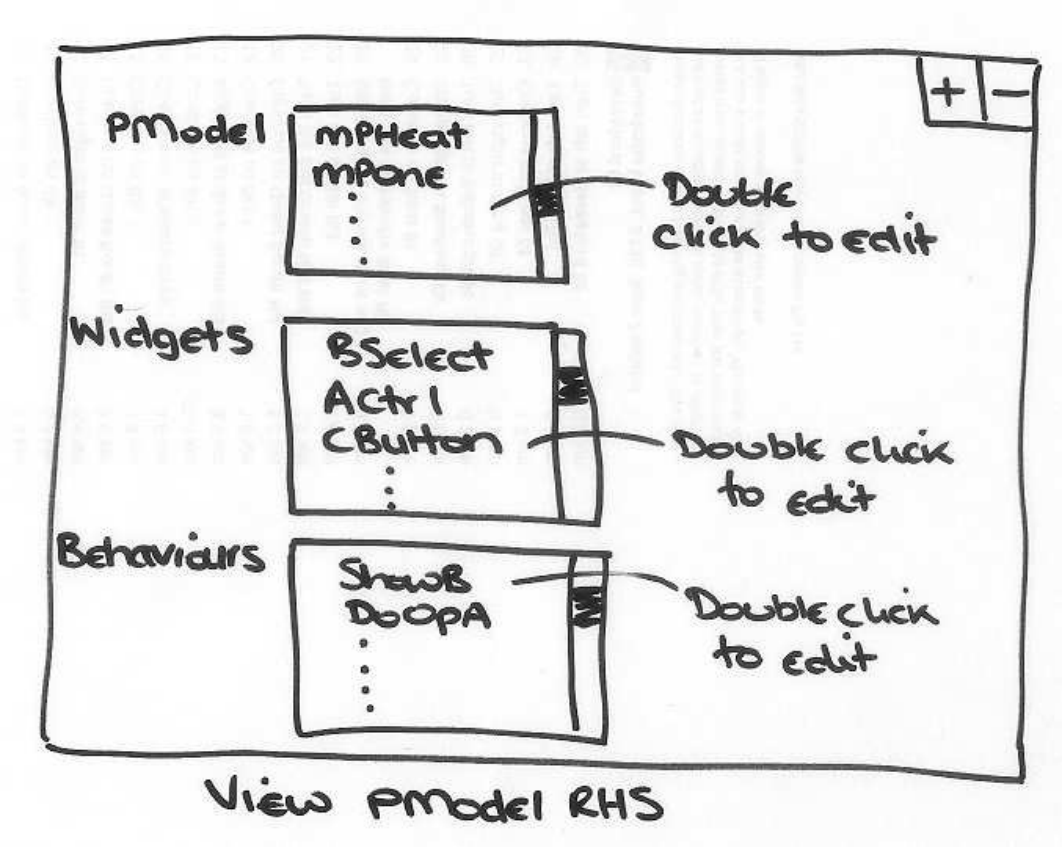


Figure 4.10: View PModel RHS Prototype

annotated with the instruction "double click to edit", but when we came to build the presentation model the designer was unsure whether the intended behaviour was to edit the *name* of the selected item (*i.e.* change the name of a selected widget) or to edit the *detail* of the selected item (*e.g.* edit the information associated with a widget name). Having to be explicit about the behaviour in order to build the presentation model meant that this ambiguity was exposed at this early stage in the design process. If left unnoticed this

could become a problem at a later stage, for example in a discussion with users the designer may describe the behaviour as being one thing but subsequently implement it as the other.

Another issue which arose as the presentation model was created was that several of the prototypes were unnecessarily complicated (making it difficult to model them). As we described them formally within the model it became apparent that they could be simplified. Similarly, the number of windows and dialogues was reduced during the modelling process as we found that in several of the dialogues all of the behaviours were also all found elsewhere in the UI or had behaviours which were more appropriate if located in other parts of the UI. These are examples of how reconsidering the UI design in a different way, *i.e.* as a presentation model, made us think differently about our proposed solution. The changes made in light of this led to amended designs which had all of the functionality of the originals but were less complicated.

4.3.6 Limitations of Presentation Models

The presentation model provides information about the possible behaviours of a UI design by describing the behaviours of each of the individual widgets. It is then possible to examine this set of behaviours to ensure that all of the user requirements are being met and also that there is a consistency between the UI design and the underlying system specification (via the *PMR*). Where a UI consists of a number of different windows and/or dialogues each part can be described in a component presentation model and then subsequently conjoined to provide information about the total UI. As such we interpret the presentation model as describing all possible behaviours of a UI design if it were instantiated as part of a real system.

There is, however, an element of UI behaviour which is missing from the presentation model and that is the dynamic behaviour of the UI itself. Whilst the *I_Behaviours* of widgets provide some information about what behaviour is available for a user to interact with the UI itself, for example by navigating

to different windows or by changing elements of the UIs appearance, this is not enough to provide certainty that the behaviour we expect of the UI is actually available to users. Consider the UI design of figure 4.11 for example, which gives a multi-window prototype for the Shape application. If we examine the overall behaviour of the complete UI (obtained by the StartWin : CircleWin : SquareWin : TriangleWin conjunction) we might initially be satisfied that the prototype has all of the required behaviours.

$$Behaviours(ShapeApp) = \{Quit, S\_ShowCircle, S\_ShowSquare,$$
$$S\_ShowTriangle, I\_OpenCircWin, I\_OpenSquareWin\}$$

Using these behaviours we can build a *PMR* which is the same as that given for the design of figure 4.4.

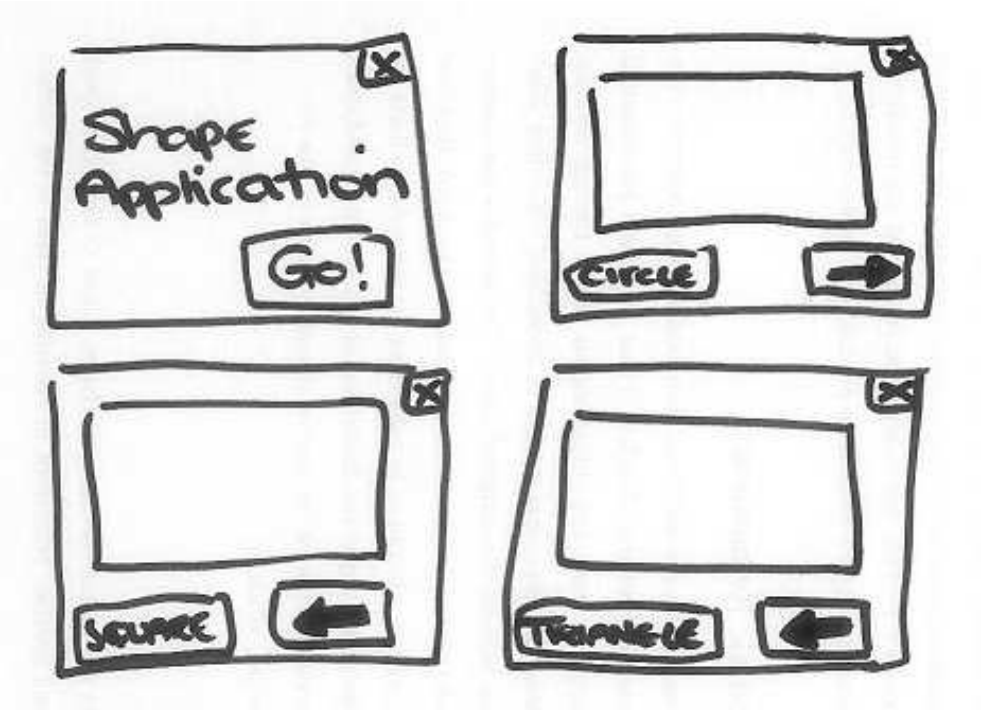


Figure 4.11: Multi Window Shape Application UI

$S\_ShowCircle \leftrightarrow SelectCircleOperation$

$S\_ShowSquare \leftrightarrow SelectSquareOperation$

$S\_ShowTriangle \leftrightarrow SelectTriangleOperation$

Multi − Window Shape Application PMR

This indicates that there is a consistency between the UI design and the system specification, but does not highlight a problem with the design, the fact that some of the behaviour is unreachable. Although the prototype has a behaviour enabling a user to show a triangle, the window that this behaviour is located in is not accessible. The presentation model, therefore, does not provide information about dynamic behaviour of a UI or the connectedness of the various parts of the UI which describes the ability of a user to navigate to different parts and access all of the provided behaviours. We cannot be sure that our design is in fact correct (by which we mean provides the necessary behaviours) unless we also consider this property. UIs are not static and so a way to represent the movement between screens, dialogues *etc.* needs to be found. In paper-based prototypes this can be achieved via story-boards or by the use of a human actor to control prototypes during an interaction description session. Also computerised prototyping tools exist which allow designers to quickly animate their prototypes in this way, for example SILK [57]. This ability to navigate through a UI in order to reach different behaviour is called reachability.

This property of reachability within a UI is important and is addressed in much of the work described in chapter 2 which deals with models of UIs. In order to provide the ability to consider reachability within our work we next present a second model, the presentation and interaction model, which uses presentation models as its basis and which provides information about dynamic UI behaviour.

4.4 Presentation and Interaction Models

In order to consider the dynamic activity of a UI (by which we mean the way a user can navigate to different parts of the UI) we need some way of considering the *I_Behaviours* which enable this navigation and how those *I_Behaviours* are linked to different parts of the UI. Rather than trying to extend the presentation model and risk making it more complicated (and therefore more of a burden and less attractive for designers to use) we instead introduce another model, the presentation and interaction model (PIM) which uses a combination of presentation models with Finite State Machines (FSM).

Finite State Machines are models consisting of states and transitions (triggered by inputs). Each state represents some configuration of a system and at any given time the machine is in one of these states (that is, we know what the current system configuration is). Inputs trigger transitions from one state to another. FSM have a notion of a start state (the initial state when the machine starts up) and accepting (or final) states.

FSM have been used in a number of different ways for UI modelling and analysis of UI properties. An early example of UI modelling can be seen in Parnas' work [70] where FSM are used to model a top level view of a system and user interactions as a way of identifying potential errors. FSM have also been used as a tool for testing UIs, for example in the work of Paiva *et al.* [68] where test cases are generated from FSMs to try and identify inconsistencies within the UI model. Thimbleby makes use of FSM to consider specific UI properties such as symmetry [93] or as part of a process to generate user manuals [95].

A known problem of using FSMs (both generally and particularly with UI models) is that of "state explosion", where the number of states of the model becomes intractably large. Given the complexity of many UIs this is potentially a big problem and it is important, therefore, to consider how we use FSM in UI modelling. For example, if we consider a state in the FSM to be the current state of an application before user input, and every possible user input

as enabling a transition (either to a new state, to an error state or back to the current state) then we will run into the state explosion problem very quickly. Rather than using FSM in this way we need a more abstract approach. For example, Belli [9] uses FSM for event sequence modelling which reduces some of the state explosion problem (but still leads to large transition diagrams) which can be used to examine specific components of the UI and generate tests rather than provide an overview of the whole UI. Another possible approach is one where each state represents a system configuration where a set of behaviours (or inputs) are possible (rather than considering individual user inputs) and where the FSM remains in that state until this set of possible behaviours changes.

We already have such an abstraction, the presentation model. We can consider each component presentation model of a UI (recall that each of these represents a distinct window or dialogue of the UI) as a state of an FSM and the inputs which cause transitions between states are then those *I_Behaviours* which are used to navigate between the different parts of the UI. The presentation interaction model is, therefore, an FSM at a high level of abstraction where each state represents a presentation model which provides the lower-level meaning.

The FSM consists of:

- a finite set of states, Q
- a finite set of input labels, $\sum$
- a transition function, δ which takes a state and an input label, and returns a state
- a start state, q_0, one of the states in Q
- a set of accepting states, F, which is a subset of Q
- a relation, R, which relates states to presentation models

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

The relation, R, is between every state of the FSM and a presentation model. When the FSM is in a particular state the relation tells us which part of the UI is currently available for user interaction. The presentation model outlines each of the possible behaviours that may occur in that state, and the design which is the basis for the presentation model indicates the visual appearance of the UI in this state. Figure 4.12 shows the relationship between the different abstractions.

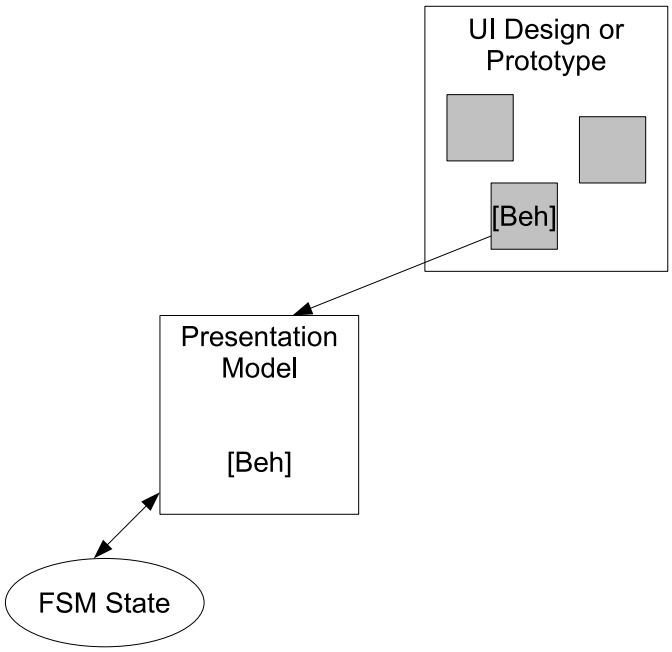


Figure 4.12: Relationship Between UI Abstractions

The start state of the PIM indicates the part of the UI which is active when the system first starts up. Accepting states are any parts of the UI which have the *Quit* behaviour in the set of behaviours of at least one of their widgets (*i.e.* it is a part of the UI where a user can exit the system). The input labels in $\sum$ are the names of *I_Behaviours* taken from the behaviour sets of the presentation models. The PIM, therefore, associates the functionality of different parts of the design with the dynamic UI behaviour which makes different parts of the interface available to the user. It provides a view of UI navigation possibilities which in turn provides a view of overall behaviour

possibilities.

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

Definition 4 *A PIM is well-formed iff it is a FSM and the labels on transitions out of any state are the names of behaviours which exist in the behaviour set of the presentation model associated with that state.*

More formally we can state:

$$WFP(Q, \sum, \delta, q_0, F, R) =_{def} \forall (q, t, q') \in \delta \bullet \exists\, b \in B[q_{PModel}] \bullet t = b$$

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

Consider another multi-window version of the Shape application the design of which is given in figure 4.13. The presentation model and *PMR* are first

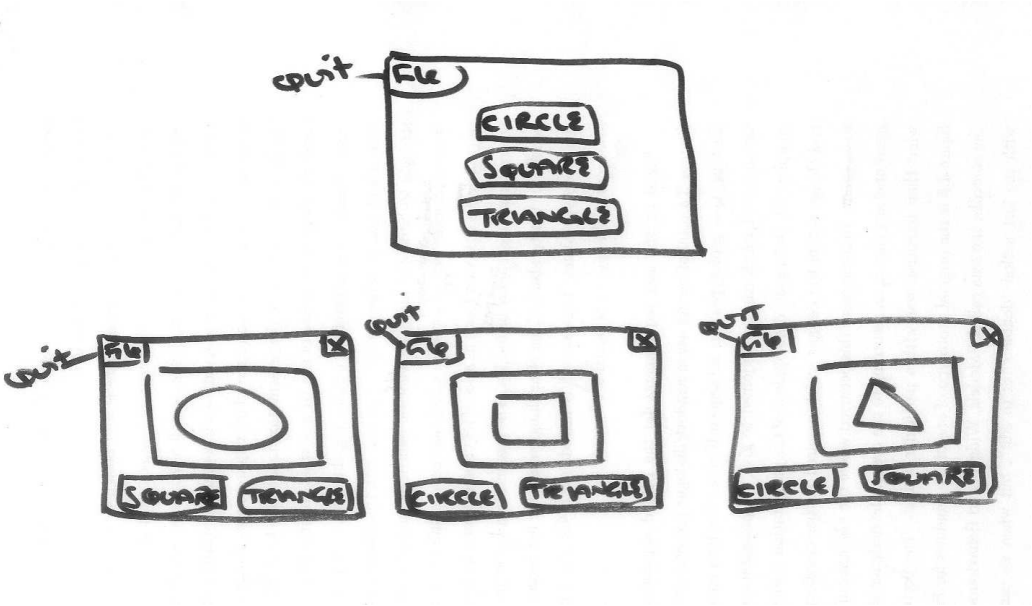


Figure 4.13: Multi-Window Shape Application Prototype

created in the usual way, giving:

PModel *ShapeApp OpenWin CircleWin SquareWin TriangleWin*

WidgetName *FMenu Quit Circle Square Triangle ShapeDisplay*

Category *Container ActCtrl*

Behaviour *Quit S_ShowCircle, S_ShowSquare, S_ShowTriangle*

ShapeApp is OpenWin : CircleWin : SquareWin : TriangleWin

OpenWin is $(FMenu, Container, ())$,

$(Quit, SValSel, (Quit))$,

$(Circle, ActCtrl, (S\_ShowCircle, I\_CircleWin))$,

$(Square, ActCtrl, (S\_ShowSquare, I\_SquareWin))$,

$(Triangle, ActCtrl, (S\_ShowTriangle, I\_TriangleWin))$

CircleWin is $(FMenu, Container, ())$,

$(Quit, SValSel, (Quit))$,

$(Quit, ActCtrl, (Quit))$,

$(Square, ActCtrl, (S\_ShowSquare, I\_SquareWin))$,

$(Triangle, ActCtrl, (S\_ShowTriangle, I\_TriangleWin))$,

$(ShapeDisplay, SValResponder, (S\_ShowCircle))$

SquareWin is $(FMenu, Container, ())$,

$(Quit, SValSel, (Quit))$,

$(Quit, ActCtrl, (Quit))$,

$(Circle, ActCtrl, (S\_ShowCircle, I\_CircleWin)),$

$(Triangle, ActCtrl, (S\_ShowTriangle, I\_TriangleWin)),$

$(ShapeDisplay, SValResponder, (S\_ShowSquare))$

$TriangleWin\ is\ (FMenu, Container, ()),$

$(Quit, SValSel, (Quit)),$

$(Quit, ActCtrl, (Quit)),$

$(Circle, ActCtrl, (S\_ShowCircle, I\_CircleWin)),$

$(Square, ActCtrl, (S\_ShowSquare, I\_SquareWin)),$

$(ShapeDisplay, SValResponder, (S\_ShowTriangle))$

Multi-Window Shape Application PMR is

$S\_ShowCircle \leftrightarrow SelectCircleOperation$

$S\_ShowSquare \leftrightarrow SelectSquareOperation$

$S\_ShowTriangle \leftrightarrow SelectTriangleOperation$

The PIM is derived by creating a single state for each of the component presentation models (which will then be used as the basis for the relation R) and creating the transitions between states based on the relevant $I\_Behaviours$. The PIM for the design above is then:

$(\{1, 2, 3, 4\}, \{I\_CircleWin, I\_SquareWin, I\_TriangleWin\},$

$((1, I\_CircleWin) \mapsto 2, (1, I\_SquareWin) \mapsto 3,$

$(1, I\_TriangleWin) \mapsto 4, (2, I\_SquareWin) \mapsto 3,$

$(2, I\_TriangleWin) \mapsto 4, (3, I\_CircleWin) \mapsto 2,$

$(3, I\_TriangleWin) \mapsto 4, (4, I\_CircleWin) \mapsto 2,$

$(4, I\_SquareWin) \mapsto 3), 1, \{1, 2, 3, 4\},$

$(1 \mapsto OpenWin, 2 \mapsto CircleWin, 3 \mapsto SquareWin, 4 \mapsto TriangleWin))$

This can be represented visually as in figure 4.14. The arrow to state 1 indicates it is the start state, and the double circles indicate final states.

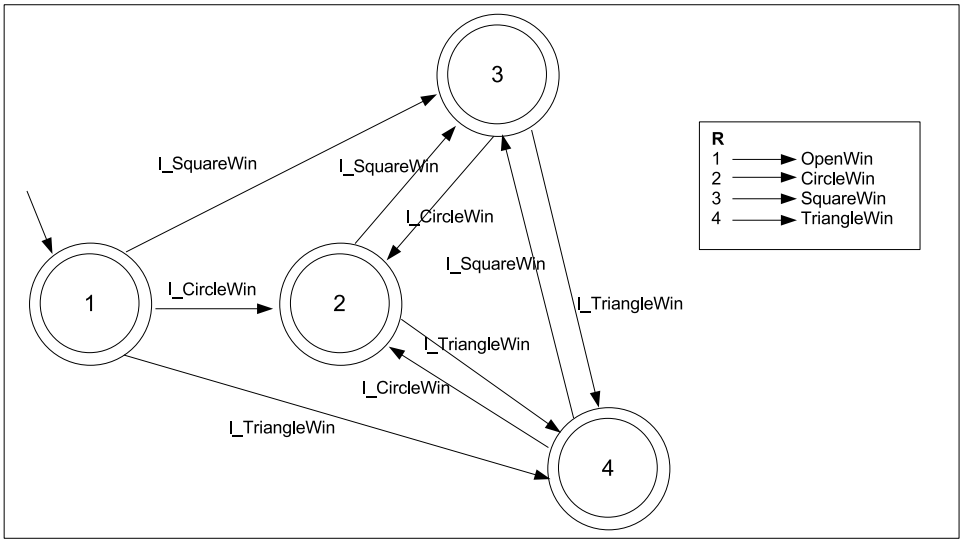


Figure 4.14: PIM for Shape Application

4.4.1 PIM Properties and Uses

The PIM can be used to check for certain properties of the UI. The first thing we are interested in is the property of reachability, that is we want to be sure that the behaviours described in the presentation models are actually available to a user at some point in their interaction. In fact, we will consider not just reachability, but total reachability which Dix and Runciman [34] describe as:

> "the ability to get anywhere from anywhere"

Not only does strong reachability ensure that all of the behaviours can be accessed by a user, but also ensures that the UI described provides ways for a user to return back to the part of the UI they came from. This property is linked to the notion of allowing users to 'undo' particular actions (by going back to previous states) which we will discuss further shortly. Reachability within the PIM provides a double-check that the presentation model (and in fact underlying design) is correct and that all required paths through the UI have been properly considered.

Section 4.3.3 explains why we wish to avoid nondeterminism in presentation models. It is, however, possible for nondeterminism to be included accidentally and go undetected. When we build the PIM from a presentation model, however, any nondeterminism is exposed and can be corrected at that point. Figure 4.15 shows an initial part of the PIM for the PIMed example. There are two transitions from the *ViewPModel* state with the same label, that is, the behaviour *I_OpenEditPModel* can cause a transition to two different

states and is therefore nondeterministic. Building the PIM exposed the nondeterminism and highlighted an error which had not been noticed when the presentation model was created. Just as having a different view of a design (the presentation model) can highlight problems not apparent from viewing the design alone, so creating another model from the presentation model (the PIM) can highlight problems not apparent in the presentation model.

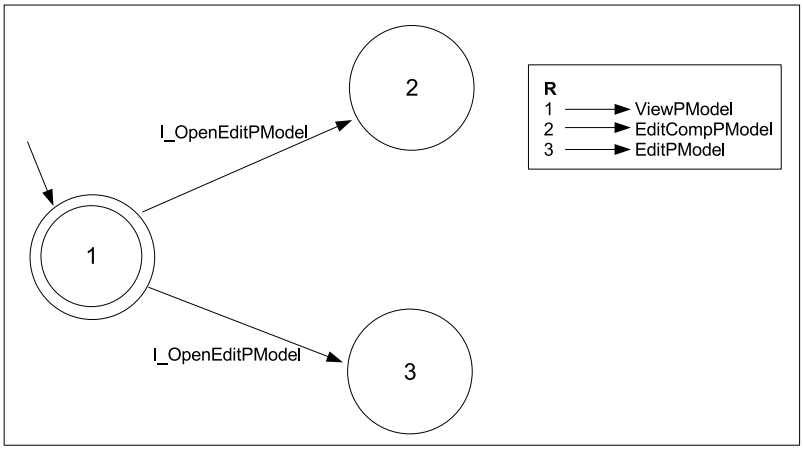


Figure 4.15: PIM with Nondeterminism

Just as the presentation model can be used to consider desirable UI properties, we can use the PIM in the same way. For example, minimising the memory load required for users to navigate through a UI is one such desirable property. Tidwell describes the cognitive cost of opening a new window or moving to a different part of the UI where a context shift is required [97]. In general, the more navigation possibilities the user has (or choices) the longer it may take them to perform tasks (based on Hick's Law which states that the time it takes to make a decision is a function of the number of available

choices [48]). The PIM gives a good indication of the complexity of the UI and its navigation by way of the number of states and transitions. If the PIM itself is difficult to navigate then we expect this will be similarly true of the UI it represents. The number of states and transitions can indicate the *amount* of navigation required by a user to perform particular actions and we can also consider the complexity of navigation via graph properties such as cycles. Consider the PIM given in figure 4.16. The cycle which exists between

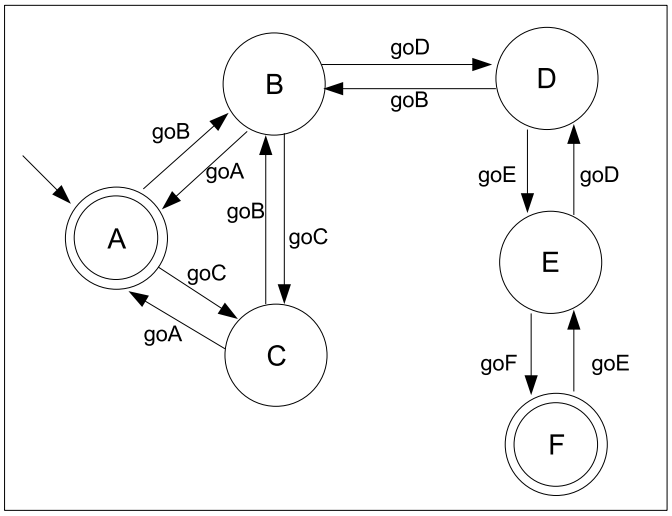


Figure 4.16: PIM with Cycle and Chaining

states A, B and C suggests straightforward navigation between these states (*i.e.* requiring a single action from the user each time), whereas the chaining between states B, D, E and F indicates increasing complexity of navigation. A user moving from state F back to state A is required to perform 4 actions, increasing their cognitive load as they must remember what these actions are. We can potentially set limits on UI navigation (based on numbers of actions we expect users to perform to reach particular states or achieve particular goals)

and then use the PIM to ensure that these limits are met. We can also use these values as metrics to compare different designs.

The ability of a user to move between particular states is linked to the concept of 'undo'. Allowing users to undo actions they have performed is often seen as desirable as it allows for error-correction or the ability for a user to try out particular actions without being committed to the result. The detail of 'undo' for every action cannot be considered by the PIM alone, but if required the behaviours of the presentation model can be paired in *do*/*undo* groups (where *do* is a particular behaviour and *undo* the behaviour which reverses its effect), this can then be used in conjunction with the PIM to ensure that *undo* behaviours are available after their respective *do* behaviours. So, for example if a *do* action causes navigation to a different part of the UI we would expect a corresponding *undo* behaviour which also returns the user to the previous UI window.

PIMs intended to provide additional information over and above our initial presentation models in order to consider informal design artefacts within a formal context. However, the fact that we can consider properties such as those given above is an additional benefit of our work and while such properties are, necessarily, limited they provide uses for the models over and above their primary purpose.

4.4.2 Visual Representation of PIMs

In our previous example we have represented PIMs both formally and visually as standard FSMs. For PIMs with a small number of states this is a suitable way of visualising them, however as the number of states increases, and the number of transitions between states also increases then the FSM view becomes more cluttered and harder to view. There is, however, another way of visually representing a PIM and that is by using the μChart language [84], [79].

Like PIMs, μcharts (which are the main entities of the μChart language) can be described using a tuple $(C, \sum, \rho, \Psi, \delta)$. C is the name of the chart, $\sum$ is the finite set of states, ρ is the start state, Ψ is a finite set of signals called the feedback set and δ is the set of transitions between states. There are some differences between this tuple and that of the PIM. In μCharts there is no notion of a final, or accepting state, there is also no designated set of input labels, rather these are implicit within the set of transitions δ (it is, therefore, possible to create such a set from δ). The name of a μchart is important, and is included in the tuple, whereas for the PIM this is not the case. The μchart tuple contains a set of feedback signals, Ψ which is not present in a PIM, and there is no relation R in the μchart tuple.

It may seem, therefore, that without a direct correspondence between the tuples of a PIM and a μchart there is no obvious reason for using this language as a visual representation for PIMs, and no immediately apparent way of how to do this. We will show next, however, that it is not only straightforward to

(easily, of course) create a correspondence between the tuples, but there are a number of benefits in doing so.

The relation, R, in the PIM is important because it enables us to understand the available behaviours within any of the states (via the related presentation model) and also provides a visual reminder of the relationship. Within μCharts there is a convention that each state name is displayed within its visual representation. So for a PIM this allows us to directly name each of the states with the same name as the presentation model it represents. If a PIM has a relation $1 \mapsto \mathit{WinOne}$ we can represent this in a μchart with a state which is called WinOne. This provides immediate visual information regarding the relationship without the need to refer to R.

Consider again the PIM for our previous example, which is:

$$
\begin{aligned}
&(\{1,2,3,4\}, \{I\_\mathit{CircleWin}, I\_\mathit{SquareWin}, I\_\mathit{TriangleWin}\},\\
&((1, I\_\mathit{CircleWin}) \mapsto 2, (1, I\_\mathit{SquareWin}) \mapsto 3,\\
&(1, I\_\mathit{TriangleWin}) \mapsto 4, (2, I\_\mathit{SquareWin}) \mapsto 3,\\
&(2, I\_\mathit{TriangleWin}) \mapsto 4, (3, I\_\mathit{CircleWin}) \mapsto 2,\\
&(3, I\_\mathit{TriangleWin}) \mapsto 4, (4, I\_\mathit{CircleWin}) \mapsto 2,\\
&(4, I\_\mathit{SquareWin}) \mapsto 3), 1, \{1,2,3,4\},\\
&(1 \mapsto \mathit{OpenWin}, 2 \mapsto \mathit{CircleWin}, 3 \mapsto \mathit{SquareWin}, 4 \mapsto \mathit{TriangleWin}))
\end{aligned}
$$

To transform this into a μchart of the form $(C, \sum, \rho, \Psi, \delta)$ we first perform a

substitution on the set of PIM state names, Q, based on the relation R:

$$(\{[1/\mathit{OpenWin}], [2/\mathit{CircleWin}], [3/\mathit{SquareWin}], [4/\mathit{TriangleWin}]\},$$

$$((1, \mathit{I\_CircleWin}) \mapsto 2, (1, \mathit{I\_SquareWin}) \mapsto 3,$$

$$(1, \mathit{I\_TriangleWin}) \mapsto 4, (2, \mathit{I\_SquareWin}) \mapsto 3,$$

$$(2, \mathit{I\_TriangleWin}) \mapsto 4, (3, \mathit{I\_CircleWin}) \mapsto 2,$$

$$(3, \mathit{I\_TriangleWin}) \mapsto 4, (4, \mathit{I\_CircleWin}) \mapsto 2,$$

$$(4, \mathit{I\_SquareWin}) \mapsto 3), 1, \{1, 2, 3, 4\},$$

$$(1 \mapsto \mathit{OpenWin}, 2 \mapsto \mathit{CircleWin}, 3 \mapsto \mathit{SquareWin}, 4 \mapsto \mathit{TriangleWin}))$$

which gives:

$$(\{\mathit{OpenWin}, \mathit{CircleWin}, \mathit{SquareWin}, \mathit{TriangleWin}\},$$

$$((1, \mathit{I\_CircleWin}) \mapsto 2, (1, \mathit{I\_SquareWin}) \mapsto 3,$$

$$(1, \mathit{I\_TriangleWin}) \mapsto 4, (2, \mathit{I\_SquareWin}) \mapsto 3,$$

$$(2, \mathit{I\_TriangleWin}) \mapsto 4, (3, \mathit{I\_CircleWin}) \mapsto 2,$$

$$(3, \mathit{I\_TriangleWin}) \mapsto 4, (4, \mathit{I\_CircleWin}) \mapsto 2,$$

$$(4, \mathit{I\_SquareWin}) \mapsto 3), 1, \{1, 2, 3, 4\},$$

$$(1 \mapsto \mathit{OpenWin}, 2 \mapsto \mathit{CircleWin}, 3 \mapsto \mathit{SquareWin}, 4 \mapsto \mathit{TriangleWin}))$$

We now use the renamed set Q as the set of states $\sum$ of the μchart.

$$(C, \{\mathit{OpenWin}, \mathit{CircleWin}, \mathit{SquareWin}, \mathit{TriangleWin}\}, \rho, \Psi, \delta)$$

The name of the PIM (which is the top level presentation model it represents) can be used as the name of the μchart giving us a value for C.

$$(ShapeApp, \{OpenWin, CircleWin, SquareWin, TriangleWin\}, \rho, \Psi, \delta)$$

The start state of the PIM q_0 is the same as the initial state for the μchart ρ and we assign the value accordingly (once again following a name substitution based on R):

$$(ShapeApp, \{OpenWin, CircleWin, SquareWin, TriangleWin\}, [1/OpenWin], \Psi, \delta)$$
$$(ShapeApp, \{OpenWin, CircleWin, SquareWin, TriangleWin\}, OpenWin, \Psi, \delta)$$

The transition function of the PIM provides almost the same information as the set of transitions of the μchart. In the PIM we describe a function over some start state and a behaviour which results in an end state. In the μchart, a transition description consists of a start and end state and signals which represent guards and actions (these signals are used to label the transition). μChart transitions consist of both input and output signals, the input signal is a guard on the transition (if the named signal is present the guard is true and the transition occurs) and the output signals are emitted as the transition occurs. PIM transitions rely on behaviours to enable the transition to take

place and as such can be considered as guards in the same manner as in μCharts. There are no output signals from transitions in PIMs, but it is acceptable for the output set of a μchart to be empty. We can then transform a PIM transition function into a μchart transition description as follows:

$$(startstate,\ behaviour) \mapsto endstate$$

$$\Rightarrow$$

$$(startstate, endstate, behaviour/\{\})$$

The states are once more renamed based on R, which gives the following μchart:

$$(ShapeApp, \{OpenWin,\ CircleWin,\ SquareWin,\ TriangleWin\},\ OpenWin,\ \Psi$$

$$(OpenWin,\ CircleWin,\ I\_CircleWin/\{\},$$

$$OpenWin,\ SquareWin,\ I\_SquareWin/\{\},$$

$$OpenWin,\ TriangleWin,\ I\_TriangleWin/\{\},$$

$$CircleWin,\ SquareWin,\ I\_SquareWin/\{\},$$

$$CircleWin,\ TriangleWin,\ I\_TriangleWin/\{\},$$

$$SquareWin,\ CircleWin,\ I\_CircleWin/\{\},$$

$$SquareWin,\ TriangleWin,\ I\_TriangleWin/\{\},$$

$$TriangleWin,\ CircleWin,\ I\_CircleWin/\{\},$$

$$TriangleWin,\ SquareWin,\ I\_SquareWin/\{\}))$$

The concept of feedback is important in μCharts as it enables communication between charts which are composed together based on transition output signals, or enables signals output from a transition to become input signals to the same chart. The set of feedback signals Ψ is, therefore, a subset of the output signals of the transition descriptions. Transitions developed from PIMs, however, have no output signals, and as such Ψ must be empty. This is not a problem as we do not require any such feedback mechanism for PIMs and it therefore makes sense for this set to be empty. An empty feedback set is a valid option for a μchart in general (not all μcharts rely on feedback) and so our final tuple describes a valid μchart (based on the syntax given by Reeve [79]).

$(ShapeApp, \{OpenWin, CircleWin, SquareWin, TriangleWin\}, OpenWin, \{\}$

$(OpenWin, CircleWin, I\_CircleWin/\{\},$

$OpenWin, SquareWin, I\_SquareWin/\{\},$

$OpenWin, TriangleWin, I\_TriangleWin/\{\},$

$CircleWin, SquareWin, I\_SquareWin/\{\},$

$CircleWin, TriangleWin, I\_TriangleWin/\{\},$

$SquareWin, CircleWin, I\_CircleWin/\{\},$

$SquareWin, TriangleWin, I\_TriangleWin/\{\},$

$TriangleWin, CircleWin, I\_CircleWin/\{\},$

$TriangleWin, SquareWin, I\_SquareWin/\{\}))$

We now have a complete, valid μchart description, but must also ensure that we have captured all of the relevant information from the PIM tuple. The relation, R, is not in the μchart tuple, but we have retained the information captured in the relation by renaming each state in the μchart with the name of the associated presentation model. The set of input labels (which are $I\_Behaviours$) is also not defined in the μchart. This information is, however, included in the set of transition descriptions. A correct interpretation of a PIM as a μchart will produce a set of transition descriptions where all of the guard labels used are in the original set of labels and *vice versa*. The only other piece of information missing is the set of final states, F. The μChart language has no concept of a final state as it is generally used to describe reactive systems capable of responding to arbitrarily long inputs. Within a PIM it is important to represent final states so that we can be sure deadlock does not occur in the UI model. That is, we always want to be sure that a user can either reach a state where they can exit the system (if that is a desirable property of the system) or some other state which may be designated as final. In systems such as kiosk applications or public interactive displays where there is no need for a quit function then within the PIM we may choose to designate all states as final as it does not matter what state the system is in when the user leaves it. Recall, however, that we have talked about reachability, and in particular total reachability, as being a property we wish to be true of our PIM. We can check for total reachability within a μchart just as we can in a FSM. If a μchart is totally reachable then we can be sure that any states which were designated

as final states in the original PIM tuple will be reachable and that there is no deadlock, we therefore have no need to specifically designate final states.

The μchart tuple we have built for the example can now be used to provide a visual representation of the PIM, this is given in figure 4.17.

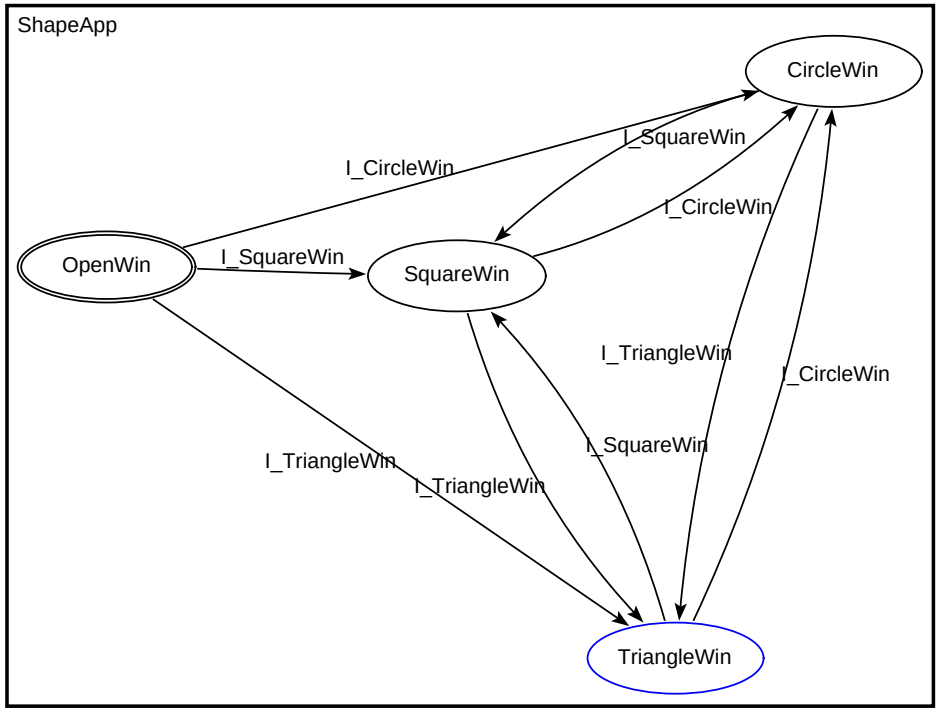


Figure 4.17: PIM for Shape Application

Having shown that we can represent PIMs as μcharts without any loss of information we also need to be sure that we preserve the meaning of a PIM when it is represented in this way. A PIM is a description of the possible behaviours of a UI design. In conjunction with the related presentation models it shows which behaviours are available to a user at any given state of the UI and how a user can move between these states. The automaton given in figure 4.14 models a system which starts up in a state where all of the behaviours show in

the *OpenWin* presentation model are available (because state 1 is the initial state and this is related to the presentation model for *OpenWin*). It is possible to make the behaviours of *CircleWin*, *SquareWin* or *TriangleWin* available by invoking the *I_CircleWin*, *I_SquareWin* or *I_TriangleWin* behaviours accordingly and moving into the related states 2, 3 or 4. It is then possible to move between states 2, 3 and 4 by again invoking the *I_CircleWin*, *I_SquareWin* or *I_TriangleWin* behaviours. All of the states are accepting states as each of the related presentation models contains a *Quit* behaviour.

Having determined the meaning of the PIM we now consider the meaning of the μchart representation of the same PIM to ensure this captures the same meaning. We consider the meaning of a μchart via the step semantics given by Reeve [79]. This provides a transition model for a sequential μchart (the individual component μcharts such as those we have presented in our PIM representations) which is a description of each of its possible transitions as an operation schema given in the Z specification language. Each of these transition schemas is then combined using schema disjunction to create one schema that describes the behaviour of the chart at each step, that is, one of the disjunct schema operations will occur at each step. This step semantics incorporates the notion of an abstract global clock where each tick of the clock represents the start of one step. There is, of course, no requirement for such a global clock for a PIM, we are interested in the possibilities of the system overall rather than at regular intervals. However, the step semantics does not alter the meaning of the PIM, and, more importantly, may be useful if we

want to consider what the state of the PIM is after a series of possible actions (at an abstract level we might consider user interactions creating a queue of behaviours and at each step one of these behaviours is actioned within the system). The fact that the interval between steps (or clock ticks) is not defined, or regular, is not important.

We can now examine the meaning of the μchart given in fig. 4.17 in light of this step semantics. There are nine transitions described in this μ-chart, therefore, at each step exactly one of the following will occur:

1. The μchart is in state *OpenWin* and the signal *I_CircleWin* is in the input set, the chart moves to state *CircleWin*

2. The μchart is in state *OpenWin* and the signal *I_SquareWin* is in the input set, the chart moves to state *SquareWin*

3. The μchart is in state *OpenWin* and the signal *I_TriangleWin* is in the input set, the chart moves to state *TriangleWin*

4. The μchart is in state *CircleWin* and the signal *I_SquareWin* is in the input set, the chart moves to state *SquareWin*

5. The μchart is in state *CircleWin* and the signal *I_TriangleWin* is in the input set, the chart moves to state *TriangleWin*

6. The μchart is in state *SquareWin* and the signal *I_CircleWin* is in the input set, the chart moves to state *CircleWin*

7. The μchart is in state *SquareWin* and the signal *I_TriangleWin* is in the input set, the chart moves to state *TriangleWin*

8. The μchart is in state *TriangleWin* and the signal *I_CircleWin* is in the input set, the chart moves to state *CircleWin*

9. The μchart is in state *TriangleWin* and the signal *I_SquareWin* is in the input set, the chart moves to state *SquareWin*

10. None of the above transitions occur and the μ-chart remains in its starting state[4]

The initial state of the μchart is *OpenWin*, therefore we know that at the first step only the first, second or third possibility given in our transition list can occur. If we compare this to the meaning of the PIM we see that we start in state 1, and therefore the first possible transitions are the same as those of the μchart. Once the μchart is in the *CircleWin* state then transitions 4 and 5 are possible, which matches the behaviour of the PIM where state 2 has the same two possible transitions. Similarly when the μchart is in the *SquareWin* and *TriangleWin* states, the possible transitions are the same as those of states 3 and 4 in the PIM.

Just as we have described the meaning of the PIM as the possibilities of the UI which are dependent on the current state and behaviour, so we have described the μchart in the same way. At each step one, or more, of the

[4]For now we assume that when none of the described signals is present the chart does nothing. We discuss this further in chapter 5.

transitions are possible depending on input. We have shown, therefore, that the use of μCharts to describe the PIM has not only preserved the meaning of the PIM, but the additional structure of the step semantics also enables us to determine additional considerations (such as how many actions, or steps, it would take to make a behaviour available for a user from a particular state).

4.4.3 Benefits of Using μCharts for PIM Visualisation

Having described how it is possible to display a PIM using μCharts and also shown that the original meaning of the PIM is preserved when we do this, we now explain what the benefits are of representing PIMs in this way.

We previously stated that if the number of states in the PIM was small then the FSM view was suitable, but that it became a problem once the number of states and transitions increased. So far we have not shown how μCharts solves this problem as we have the same number of states as in the original view. However, μCharts provides a syntactic convention called *decomposition* which allows us to embed μcharts within the states of other μcharts. This gives a hierarchical way of viewing the chart either at a high-level where a single chart contains decomposed states, or as several *decomposed* charts, rather than having to view the whole chart in one diagram. So, a single PIM can be viewed as a number of decomposed μcharts where the original meaning is preserved (via the μCharts semantics).

As an example we return to the PIMed case study. Recall that the presentation model for the total application UI design for this system has twenty

seven different presentation models. The PIM, therefore, has twenty-seven different states and some of these states have more than ten incoming and outgoing transitions. In terms of presentation model composition this might be described as:

$PIMed\ is\ w1 : w2 : w3 : w4 : \ldots : w27$

so that the complete UI for the application is the composition of each of the twenty seven windows. However, suppose we create a more hierarchical composition for the presentation models, such as:

$PIMed\ is\ w1 : w2 : w3 : w4$

$w1\ is\ w5 : w6 : w7$

$w2\ is\ w8 : w9 : w10 : w11 : w12 : w13$

$w3\ is\ w14 : w15$

$w4\ is\ w16 : w17 : w18 : w19$

$w14\ is\ w20 : w21 : w22 : w23 : w24$

$w15\ is\ w25 : w26 : w27 : w28 : w29 : w30$

Notice that we have increased the number of component presentation models now as we have created additional compositions, however the presentation model semantics ensures that the intended behaviours are correctly preserved. As was explained in section 4.3.1 the composition operator : creates the union of the sets of behaviours from the component models so that all behaviour is preserved (and not increased) by such a restructuring.

Now we can use decomposed μcharts to display these compositions. Figures 4.18 and 4.19 show two of the actual μcharts for the PIMed application.

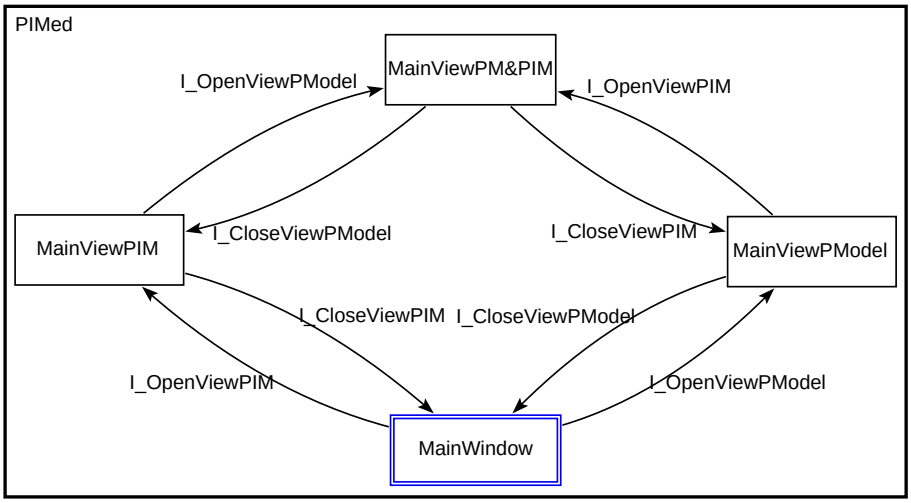


Figure 4.18: High Level PIM for PIMed

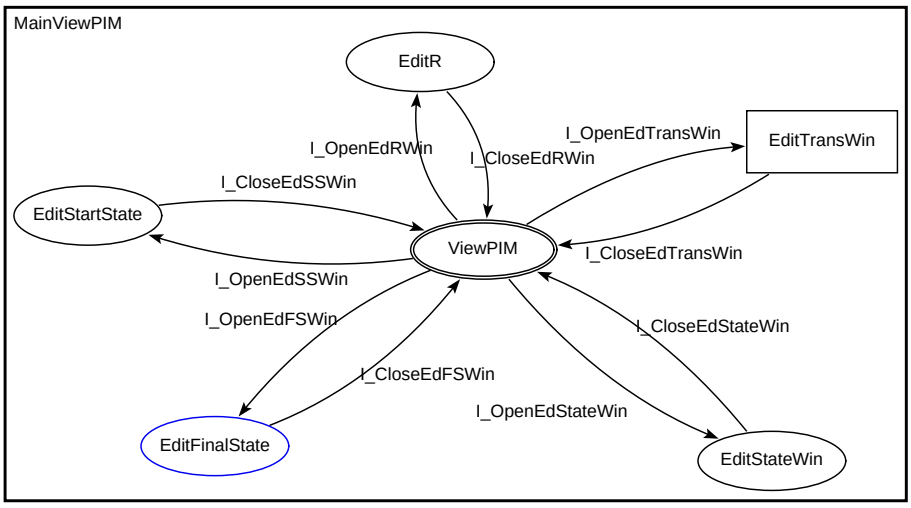


Figure 4.19: Part of PIMed PIM Decomposition

The first of these, figure 4.18, gives a high-level view of the PIM and consists of four decomposed states. Part of the PIMed presentation model describing this would appear as:

PIMed is MainWindow : MainViewPIM : MainViewPMandPIM : MainView-PModel

Figure 4.19 then provides the detail of one of these decomposed states, *MainViewPIM* and is itself based on part of the presentation model which is: *MainViewPIM is ViewPIM : EditR : EditTransWin : EditStateWin : EditFinalState : EditStartState*

When a μchart is in a state which is a decomposed state, the embedded chart is considered active, and is in the start state of the decomposition. In this example if the chart in figure 4.18 is in the *MainViewPIM* state, then all of the behaviours of the *ViewPIM* state (the start state of the *MainViewPIM* decomposition) are available to a user.

Another benefit of using μcharts to represent PIMs is that we now have an additional way of considering the meaning of the model via the Z semantics of μCharts. An example of how this is helpful is the ability to use Z theorem-proving tools to prove properties about PIMs via the underlying Z of the μchart which represents it. There are freely available tools (AMuZed and ZooM [6]) which enable us to create and edit μcharts and then produce the Z representation of those charts. The Z can then be used with any Z thereom-prover, such as Z/EVES [83] for example. Suppose we want to be sure that several different sequences of user actions all lead to a state where the same behaviours are possible; rather then having to manually check for this via inspection of the PIMs (which becomes increasingly difficult as the UIs become more complex) we can use a theorem-prover. We have already discussed how we can use presentation models to consider equivalence of designs. We can similarly now consider equivalence of PIMs (and again check correctness of

this using theorem-proving). Having shown that two presentation models have behavioural equivalence we can then ensure that they are also equivalent in sequences of behaviours.

As well as the decomposition syntax available within μCharts there is also a composition operator which allows two separate μcharts to be composed together into a single system where both parts of the chart are able to communicate with each other on sets of signals. In our next chapter we will show how this enables us to bring together models of the UI and models of the underlying system into a single representation and how we are then able to use this to consider refinement.

4.5 Proving Properties of PIMs

We have stated that one of the advantages of using μcharts as a representation for PIMs is that the underlying Z semantics of the charts allows us to prove properties about the models. We next present some examples of this using the PIM given in figure 4.17 and the Z/EVES theorem prover. Using the chart translation tool, ZooM [6] we have taken the μchart of the PIM and translated it into its underlying Z. We give the Z for the chart in full in Appendix D. Now we are able to use this in conjunction with Z/EVES to check for properties of the PIM, which in turn means we can check properties of the UI design modelled by the PIM.

A useful property to check for is that of reachability which is, as we have

discussed, an important criterion for our models. Suppose we wish to check the PIM of figure 4.17 for total reachability, that is, is it possible to reach any state of the PIM from any other state? We can check this by testing for the following:

- Starting in OpenWin is it possible to reach CircleWin?
- Starting in OpenWin is it possible to reach SquareWin?
- Starting in OpenWin is it possible to reach TriangleWin?
- Starting in CircleWin is it possible to reach SquareWin?
- Starting in CircleWin is it possible to reach TriangleWin?
- Starting in CircleWin is it possible to reach OpenWin?
- Starting in SquareWin is it possible to reach CircleWin?
- Starting in SquareWin is it possible to reach TriangleWin?
- Starting in SquareWin is it possible to reach OpenWin?
- Starting in TriangleWin is it possible to reach CircleWin?
- Starting in TriangleWin is it possible to reach SquareWin?
- Starting in TriangleWin is it possible to reach OpenWin?

So, for example to check whether we can reach CircleWin from OpenWin we submit the following to Z/EVES:

```
try ShapeAppSys [\cShapeApp := \ShapeAppOpenWin, \cShapeApp' :=
                                                    \ShapeAppCircleWin];
```

This attempts to prove the conjecture given, that performing the *ShapeAppSys* operation with the chart in a start state of *OpenWin* leads to an end state of *CircleWin*. The *ShapeAppSys* operation is the disjunction of all possible transitions of the chart. The test then checks if there is any transition (or series of transitions) where the chart starts in the *OpenWin* state and finishes in the *CircleWin* state. If this is possible then we will know that such a transition can be made by the chart and that the *CircleWin* state is therefore reachable from the *OpenWin* state. Z/EVES provides the following response:

Proving gives...

$$? \in \mathbb{P}(\{I\_Circlewin\} \cup (\{I\_TriangleWin\} \cup (\{I\_SquareWin\} \cup \{I\_CircleWin\})))$$

$$\wedge ! = \{\} \wedge$$

$$(\exists\, active\_ : \mathbb{P}\, \mu_{State} \bullet (ShapeApp \in (active\_) \wedge I\_CircleWin \in ?))$$

If we examine each part of this response we can interpret the answer Z/EVES has provided. The first line states that ?, which is the set of input signals, is one of the power sets of $\{I\_Circlewin, I\_TriangleWin, I\_SquareWin\}$. This is expected as these are all of the allowable inputs for the chart, so any input will one of its subsets. The second line $! = \{\}$, requires the set of output signals to be empty. Again this is expected as the chart has no outputs on any

of its transitions. The line $\exists\, active\_ : \mathbb{P}\, \mu_{State} \bullet (ShapeApp \in (active\_)$ is the requirement that the chart in question, that is *ShapeApp*, is currently active (in the case of single charts with no decompositions this is always true). The last part of the conjunction, $I\_CircleWin \in?$ states that the signal $I\_CircleWin$ is in the set of input signals. So, it is possible for the chart to start in the *OpenWin* state and finish in the *CircleWin* state via a transition which has the signal $I\_CircleWin$ as its guard if the signal $I\_CircleWin$ is in the set of input signals.

We can repeat this process for each of the other state pairs and similarly show that each state can be reached. Of course, this chart does not have total reachability. It is one of the original requirements given that we can never return to a state where no shape is showing once we have displayed a shape. As such it should not be possible to reach the *OpenWin* state (the only state where no shape is showing) from any other state. We can test this in exactly the same way.

For example we can try and prove the following:

```
try ShapeAppSys [\cShapeApp := \ShapeAppCircleWin, \cShapeApp' :=
                                                    \ShapeAppOpenWin];
```

In this case Z/EVES returns:

```
false
```

which shows us that the model is correct (assuming we can perform the same proofs from the start states of *TriangleWin* and *SquareWin*).

For small systems, such as that given in our example, we can perform these reachability tests manually by inspecting the charts, however for larger examples (and especially those containing decompositions) this is not so straightforward. In these cases the ability to use theorem-provers rather than manual methods are an additional advantage of using μcharts as a representation for PIMs.

4.6 Discussion

In this chapter we have introduced two models, the presentation model and the presentation and interaction model (PIM). These models form the basis of the first step in integrating informal design artefacts and a formal system development process. The presentation model provides a structured way to describe informal design artefacts, such as prototypes, in a lightweight, easy-to-use manner. A relation can then be created between the behaviours described in the presentation model and the operations of a formal system specification which provides a method of checking for consistency between the two design streams. If the *S_Behaviours* of the UI model can be related to operations of the specification then it shows a correspondence between the expected behaviours of the overall system.

Presentation models provide a static view of UI designs by describing all possible behaviours of categorised widgets, this can then be extended to include dynamic properties of UIs in the PIM which provides a view of navigational

possibilities of the UI. The PIM can be used to check for total reachability and deadlock within the design. The combination of presentation model and PIM allows us to ensure that not only do the expected behaviours exist in the UI (based on user requirements and system specification) but also that the behaviours are accessible by users during their interaction with the system.

The models do not rely on UI designers giving up their existing practices, but rather are aimed to work in conjunction with their existing methods and in particular with the UCD technique of prototyping. That is we expect that all of the activities they would normally undertake during the UI design process (such as task analysis, heuristic walk-throughs, usability testing *etc.* are still undertaken as usual. As such, the models provide a bridge between the UI design process and the formal specification process by formalising those aspects of the design which may otherwise be ambiguous. By formalising the behaviour of UI designs the models, in conjunction with the prototypes they are based on, provide a full description which does not rely on the designer describing what their intentions were. This ensures that information does not get forgotten, or changed arbitrarily over time. Rather than replacing existing design artefacts then, the models are coupled with them and used to support the design. This means that we do not need to formalise visual aspects of the UI as they already exist in the design artefacts, we formalise only what is useful for the formal process but everything else is retained in its original form. The models, therefore, preserve the meaning of the UI while the prototypes preserve the visual considerations and there is no loss of information.

The original aim of developing the models was to create a way of linking the informal UI design process with the formal specification process at the point where a completed specification is available and UI designs and prototypes have been developed: this aim has been achieved. In addition, we have also shown that using the models may provide benefits to the UI design process over and above these original intentions. The act of creating the models from design artefacts forces designers to consider the UI from different points of view, and as such can expose previously hidden problems, or weaknesses, in the UI design. We can also use the models to consider desirable properties of UIs at an early stage in the design process. Aspects such as reactivity of a UI or consistency can be difficult to consider until designs have become more concrete. The models here provide the ability to check for these properties at the prototyping stage which means changes can be made more easily if problems are found.

The formal models described provide a way of comparing proposed system behaviour with proposed UI behaviour, but they also allow for comparing designs themselves, via the equivalences we have designed. This is a useful property when considering refinement for UIs as we will show in chapter 6. Additionally the use of μCharts as a medium for visually representing PIMs provides the first step in bringing together formal representations of UI and system into a single description: we discuss this in the next chapter. Having first shown how we can build the FSM for a PIM and subsequently transform that into a μchart it is now clear how to directly represent the PIM as a

μchart (that is, we do not need to first create the FSM and then convert it into a μchart).

Now that we have the basis to interpret designs formally and a link between specifications and prototypes we can move on to our next consideration, that of correctly transforming the specification and the design into a single, implemented system. In the next chapter we begin by proposing a method of combining the two parts of the design, the system and the UI, using the μCharts language.

Chapter 5

Composition of System and UI

5.1 Introduction

We have so far shown how we can formally describe UI designs (by way of presentation models and PIMs) and used these models to develop a link between the sorts of artefacts produced during a UCD process and a formal system specification. This link is formalised by the presentation model relation (*PMR*), and both this and the models themselves provide a number of benefits to the design process. One of the motivations for developing this link was to be sure that there is consistency between the aims of UI designers and system designers: that is, to be certain that they are working towards the same system.

Another important concern within our work is that of ensuring that the final implementation (based on these early designs) maintains the correctness which has been proven earlier using the presentation model and *PMR* as we

have discussed in chapter 4. We have discussed how the separation of system and UI can lead to problems at implementation stage, and using the formal models of design artefacts and developing the *PMR* are the first step in avoiding such problems.

Before we can move on to considering methods of implementation (via refinement) for both UI and system, we first need to consider how to bring together our descriptions of the system and UI. Although we now have a formal model for them both, as well as a link between the models (via the *PMR*), the formalisms used for each are still different: our UI is described as a presentation model and PIM, while our system is still described by its formal specification. Our next step is to describe a way of bringing the two parts together into a single model, and use this as the basis for the next stage of development. This single model can be considered the joining together (or composition) of the system and UI and we will denote it by $(UI \mid\mid Sys)$. The two parts of the composition may be at any level of abstraction between prototype/specification and implementation, and the two parts may be at different levels of abstraction from each other. As long as certain criteria are met (which we discuss next) we say that the composition models a working application. That is, if implemented it would produce a fully working system and user interface. This single model describes not only each part of the application, but also details the communication between the two, so it should show how the system and UI communicate via particular behaviours and what effect they have on each other.

One way of producing a single description of system and UI in a common formalism would be to take the presentation model and expand this into a formal specification itself (for example we could take each of the widget tuples and describe these using Z). Then, through a process of merging the two specifications (system and UI) we could expand the combined specification to describe the links between the widgets and their behaviours and the operations of the system. This would involve finding the common operations (those described by $S\_Behaviours$) and explicitly modelling the interaction. However, we would also need to consider the availability issues we have captured in the PIM (how the change in UI state affects the availability of behaviours) and find some way of also including UI navigation within the specification. Not only does such an approach produce a long specification (describing each widget of the UI is a lengthy process in itself) which is made harder to understand by the detail of the interactive behaviours, but there is little to be gained from such an approach. There is, in fact, a better way of describing $(UI \parallel Sys)$ based on the PIM and the fact that we have a method of modelling PIMs using μCharts.

Once we have a common model for $(UI \parallel Sys)$ we will show that there are conditions needed to ensure that the model is of an interactive application. We have previously discussed how links between presentation models and system specifications provide a way of checking for consistency within the design process, and once we have a single $(UI \parallel Sys)$ model we can extend this further and develop a notion of *valid* applications. That is, we show that

composing just any (possibly unrelated) system and UI pair will not satisfy the requirements which ensure that a correctly implemented application can be developed from such a model. Once we have described what these requirements for validity are, we will show how they can be incorporated in a single (UI || Sys) model and how this is represented using μcharts.

5.2 Validity

In order to define what we will call a valid application, which is a (UI || Sys) pairing which meets certain correctness conditions, we must consider what these conditions are. It is possible to put together any system description and UI model (irrespective of their underlying requirements) but we want to do so only when they are intended to be parts of the same application, in which case we want to be sure that they are designed to interact with each other correctly. That is, both parts should be designed with the same intended interactive behaviours so that the UI provides a way for a user to access the functionality of the system. This notion of correct interaction is based on the consistency principle we have in section 4.3.5. We have shown how we can create a relation (the PMR) between a presentation model and a system specification which ensures that they have the same underlying behaviour (or at least a common set of behaviours). We now extend this idea to consider the nature of the interaction between system and UI and the effect this has when we compose the two models together to produce (UI || Sys).

Firstly the UI and system pair we are considering must be interactive, that is there must be at least one related system operation and UI behaviour. This ensures that there is some connection between the two which allows a user to interact with the system via the UI (as exemplified by the diagram presented earlier in figure 4.3). We use the relation between the UI and system models to define both a consistent design approach (in terms of the end goal) and the ability of the two parts of the application to interact with each other. However, we now extend the notion of consistency.

Just as we have divided the behaviours of the UI into *I_Behaviours* and *S_Behaviours*, we can similarly consider the operations of the system specification based on whether they describe operations which should be available to users and those which relate to underlying system behaviour. Those operations which should be available to users are related to the *S_Behaviours* of the UI in the *PMR*. At the moment our requirements on the relation, that it is a total, many-to-one relation do not prevent a situation where operations in the system specification which should be made accessible in the UI are not related to any *S_Behaviour*. For example, consider the following relation:

$$S\_DrawCircle \mapsto DrawCircleOperation$$

$$S\_DrawSquare \mapsto DrawTriangleOperation$$

$$DrawSquareOperation$$

The relation is a total, many-to-one relation as required for mutual inter-

action, but there is a mismatch between the intentions of the UI and the system. The unrelated *DrawTriangleOperation* is not an underlying system function, but one which should be made available via the UI. If we describe those operations which should relate to UI behaviours as S_Operations, and those which describe underlying functionality only as F_Operations then consistency also requires that all S_Operations of the system specification are related to *S_Behaviours* of the presentation model.

Definition 5 *A* valid *application is comprised of a system and UI pair such that the* PMR *between the presentation model of the UI and the system specification is a total, many-to-one, onto relation.*

5.3 Formalising the Composition

Now that we have a definition of what a valid application is, and what the requirements are on system and UI pairs to ensure their composition is valid, we describe how we can formalise this composition using μcharts. We have already shown how the PIM of a UI design can be visualised using μCharts. The μcharts we have seen so far are individual charts (called sequential charts), but the language also contains a composition operator which allows us to put two sequential charts together and define how they can communicate with each other as well as with the external environment.

μcharts are used to describe reactive systems, that is, they describe a system which responds to some external stimulus, which is called the environment,

and can produce outputs back to that environment. If we consider a UI as a reactive system, then we can consider the external environment to be the user. Inputs to the chart come from the environment. So, in our case where we consider the user as the environment, input signals are the behaviours which the user can invoke by interacting with the widgets of the UI. Recall that in the presentation model the widgets have behaviours associated with them: if a user interacts with a widget which has a behaviour B, then B would be a signal input to the chart. We have already shown PIMs modelled as μcharts and we will use this model (the PIM) to represent the UI in ($UI \parallel Sys$). Although the PIM is an abstraction of the full UI it has all of the behaviours of a fully implemented version of the same UI (via the related presentation models), and as it is the behaviours we are interested in the PIM is therefore a satisfactory representation within the composition.

In order to describe both UI and system as composed μcharts we must also model the system itself as a μchart. Up to this point there is no requirement for any particular specification language to be used for the underlying system, but in order to consider both parts of the application together (which will allow us to investigate refinement properties among other things) we must now model the system as a reactive system which then allows us to model it as a μchart.

5.3.1 Semantics of μCharts

One thing we do need to consider when using μcharts is the choice of semantics. As we have previously stated, μCharts has semantics given in Z, but in fact

there are four subtly different versions of the semantics. These semantics are described fully in [80] and are called *do-nothing*, *partial chaos*, *total chaos* and *firing conditions* semantics. Each of these provides a slightly different interpretation of chart behaviour when signals which trigger transitions are not present.

In our previous chapter we described PIMs following the *do-nothing* semantics, that is we assumed that if a PIM is in a particular state and there is no signal input which triggers a transition then nothing happens. While this was convenient for our initial description of PIMs as μcharts it is in fact not the most suitable choice of semantics. We next give a brief description of each of the semantics and explain why this is so, and the implications of the choice we make.

The *do-nothing* semantics can be described as a μchart ignoring any input which does not trigger a defined transition in its current state. That is, it is not just that no signal is present, but that any signals that are present are ignored as they have no defined behaviour and so we assume the chart does nothing. This appears to model a well-behaved system where any accidental or incorrect input from the user, for example, will not cause anything untoward to happen.

It is however, a restrictive model which does not allow for under specification (which is often useful for our system models where we use nondeterminism to delay decision making early in the design). To understand why this is so consider the two charts shown in figure 5.1. The chart on the right explicitly

models the do-nothing behaviour of the chart on the left. When the chart on the right is in state X input of signal a will cause the transition to state Y and output of signal b. Once the chart is in state Y it will remain there (the guard given as / is equivalent to *true*). However it may be that we subsequently want the chart to do something when it is in state Y and signal a is input, but as yet we have not defined what that behaviour will be. Under the *do-nothing* semantics that behaviour is defined (on input of signal a the chart remains in state Y) rather than left undetermined. As we will see later when we consider refinement, this is not necessarily the best option as it prevents such nondeterminism in system charts and restricts the permissible refinements.

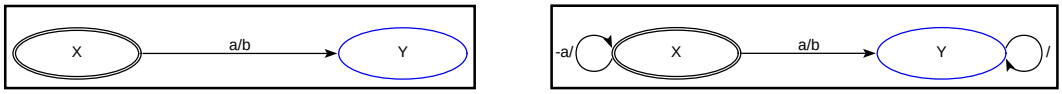


Figure 5.1: Do Nothing Behaviour Explicitly Defined

The *firing conditions* semantics does not ignore undefined behaviour, but rather treats it as a signal to terminate. If a chart is in a certain state and a signal is input which does not have a defined transition from that state then termination results. As an implemented system this is certainly not a suitable option as it would be analogous to a system which quits every time an unexpected input is seen. For example suppose we have a UI where a user can print something using a keyboard shortcut, but this is only defined within certain windows of the UI. If the user then provides this input where it is not defined the system would quit.

It is possible to adopt this semantics and include explicit signals (as we

shall see shortly) to prevent such a termination ever occurring, in which case error conditions are included in the model forcing designers to consider them when transforming the model to an implementation. However, rather than assuming termination when an error occurs (which may be a controlled way of behaving when a system does not know what to do but from a user's point of view is unlikely to be satisfactory) we will take the approach that we will define what happens in such cases.

The *partial chaos* semantics allows for nondeterminism in a model by allowing undefined behaviour to take place outside of the defined transitions. When a chart is in a particular state and an input is seen which does not have a defined transition (including empty input) then anything can happen. The semantics is called *partial chaos* because once a signal with defined behaviour does appear in the input then chaotic behaviour stops and the chart will once more behave as expected. The chaos is limited only to each step where there is undetermined behaviour based on inputs. Although we might expect that a semantics which allows nondeterminism would not be a suitable choice for our composition (as our UI models do not contain nondeterminism) it is, of course, possible for the system model to be nondeterministic and we wish to create μcharts for the system as well as the UI, and so we will not rule out this semantics on that basis. In fact, we again have a situation where the model is assuming that the system can revert from chaotic behaviour to being well-behaved once more without making any requirements on the implementation to guarantee this is satisfied. As was the case with the *do-nothing* semantics

we would rather make such behaviour explicit.

For our μcharts of UI and system composition we will use the remaining version of semantics, the *total chaos* semantics. As the name suggests, this semantics describe chart behaviour where not only does the chart behave chaotically when undefined inputs occur, but we cannot be certain that it will ever stop behaving chaotically once this has happened. This model, therefore, gives us the worst-case scenario and ensures that as designers we consider all possible inputs in each state in order to prevent such chaotic behaviour occurring. Rather than allowing the nondeterminism that chaos brings into the UI chart in the composition we will explicitly include transitions which ensure that we are never in that situation.

Consider the two μcharts shown in figure 5.2. The chart at the top con-

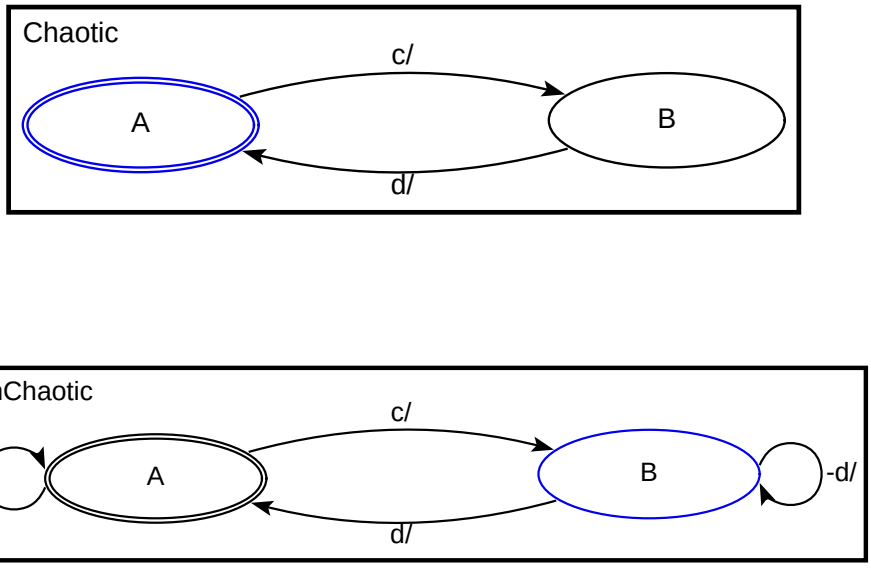


Figure 5.2: Chaotic and NonChaotic Charts

tains nondeterminism. If the chart is in state A and anything other than an

input of signal c occurs then behaviour is undefined and the chart can behave chaotically. Similarly, in state B only the input of signal d leads to guaranteed behaviour. The bottom chart, however, prevents the nondeterminism by explicitly constraining the chart to remain in the same state if the signals with defined transitions do not appear in the input. The "-" symbol before the signal in the two loop transitions has the meaning that if this signal is *not* present in the input then this transition (to remain in the same state and perform no output) must take place. It is not always the case that loop transitions (transitions which start and end in the same state) have the same meaning as *do-nothing*, as it may be the case that some output signal is emitted. However, where the input signal is the negation of any signals used in transitions out of that state and there is no output, then the meaning of such loop transitions is do-nothing. The inclusion of these loop transitions leads to the same chart behaviour as the *do-nothing* semantics but in this case the behaviour is controlled explicitly by these transitions rather than just assumed. When the model is transformed into an implemented application we therefore expect that this behaviour will be implemented as it is an explicit requirement of the model (assuming, of course, that we have some method of transforming the model into an implementation which preserves all properties of the model). We will discuss this further in the next chapter. We have then a method of under-defining our models where chaos may result from undefined inputs (which allows us to subsequently refine away chaotic behaviour) as well as a method of explicitly defining do-nothing behaviour when required.

5.3.2 Composed μcharts for System and UI Composition

μCharts provides a syntax for composing two sequential charts together as well as semantics to describe the meaning of the composition. Recall that a μchart is described by a tuple of the form $(C, \sum, \rho, \Psi, \delta)$. The composition operator || joins together two charts, and has the following syntax:

$$(C_1, \sum_1, \rho_1, \Psi_1, \delta_1) \mid \Psi \mid (C_2, \sum_2, \rho_2, \Psi_2, \delta_2)$$

Ψ is the set of signals that C_1 and C_2 can use to communicate with each other. At each step signals output from either of the charts which are in the set Ψ are also seen as inputs to each of the charts (transitions are assumed to take no time so that the consumption of input and the provision of output occur at the same time).

Our general model for $(UI \mid\mid Sys)$ is given in figure 5.3. The top part of the composition is a sequential μchart of the UI, which will in fact be the PIM, and the bottom part of the composition is a sequential μchart of the system. Each of the charts will react to inputs from the environment as well as any signals in the feedback set present at any given step. The feedback set, *UISysCom* contains all of the signals that the UI and system can communicate to each other. We have already defined how such communication might occur in the *PMR*, which shows the relation between behaviours in the UI and operations

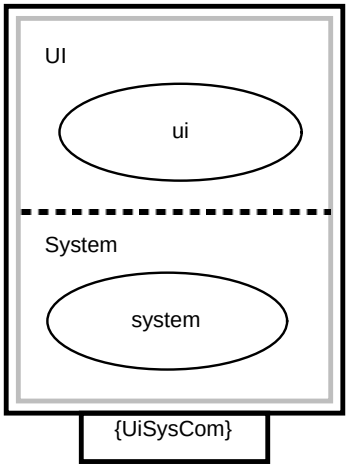


Figure 5.3: Composed μchart

of the system. Abstractly we can consider that when a user interacts with a UI (via a widget) the *S_Behaviour* of that widget uses the related system operation to communicate the user's intentions. Recall, however, that our PIM description does not currently include any *S_Behaviours* directly, these are contained within the presentation models of the related states rather than being made visible within the PIM. The reason for this is that the PIM shows us the dynamic UI behaviour, which relates to the *I_Behaviours*, and we rely on the presentation model and *PMR* to give the relationship between UI and system behaviours. For the purposes of composing the system and UI, however, we want to explicitly model this relationship and therefore need to somehow include the *S_Behaviours* explicitly in our model.

In fact this is a straightforward addition to the PIM. In the conversion from the FSM representation of a PIM to a μchart the output signals on transitions were left empty. The purpose of such signals in general is to provide output

from a chart to either the environment or to some other chart (via the feedback set in the case of composed charts). We can therefore consider *S_Behaviours* and their related system operations as transitions where an input signal (which is the name of an *S_Behaviour*) provides an output signal (which is the name of an operation of the system related to that *S_Behaviour*). This will enable a transition in a UI chart to output a signal which can be used as an input to a transition in the system chart. We will, of course, have the same requirements on transitions which use *S_Behaviours* as signals as we do for *I_Behaviours*, which is that any *S_Behaviours* on a transition out of a state of the PIM must be included in the behaviour set of at least one widget in the related presentation model.

There are two possible types of transitions containing *S_Behaviours*. The first is where the *S_Behaviour* is related to some dynamic UI behaviour, this will be the case when an *S_Behaviour* and *I_Behaviour* are in the same behaviour set of any one widget (the meaning of multiple behaviours is that *all* behaviours occur when the widget is interacted with). In this case the guard of the transition will contain both behaviours as signals (μCharts uses the syntax of "." for the concatenation of multiple signals and "+" for the disjunction of signals) and the system operation related to the *S_Behaviour* will appear as an output signal. The second type of transition is where an *S_Behaviour* is not related to any dynamic UI behaviours. In this case a loop transition will be used to indicate that there is no change of state in the UI when the *S_Behaviour* is seen as an input, but only the output of the related system

operation occurs. As an example of this in figure 5.4 we repeat the PIM first shown in figure 4.17, but this time we include the *S_Behaviours* as well as the loop transitions required to prevent chaotic behaviour.

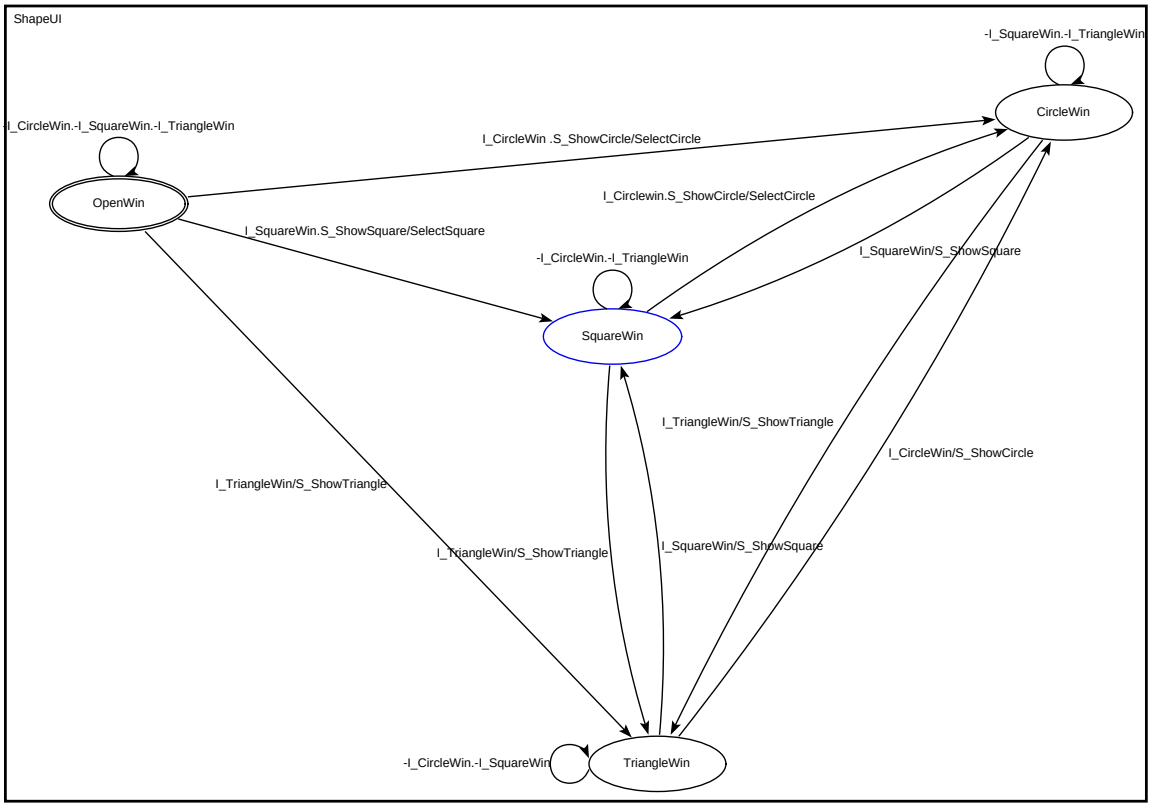


Figure 5.4: PIM for Shape Application

We stated that the environment of the chart (which provides the inputs) is considered to be the user (so that inputs are the results of their interactions), however, both sequential charts in the composition can receive inputs from the environment and we do not want the user to provide inputs to the system directly but rather to interact via the UI. The model we have given in figure 5.3 does not prevent the user from interacting directly with either part of the composition (via inputs) and so does not correctly describe our required

interactions. Fortunately μCharts provides a way to restrict both input signals coming from, and output signals going to, the environment. The mechanism for this is via the *interfaces* of a chart. Each μchart has an input and output interface which consists of a set of signals which determine which signals will be accepted from the environment and which signals can be output to the environment. The interfaces act as filters and restrict what is seen by the chart and what is output.

All μcharts have an implicit input and output interface which consists of all of the signals of its transitions. That is, if we do not make any explicit statement about the interfaces then all signals from the environment will be accepted, and all outputs from transitions will be output to the environment. As we do not want this to be the case we will use explicit interface descriptions to control inputs and outputs in a way which matches our requirements. The PIM we created for the multi-window version of the Shape application, given in figure 5.4, can be composed with a chart representing the Shape system (which is based on the specification given in chapter 3). This composition is given in the μchart of figure 5.5

We denote this composed chart by the following expression:

$ShapeUI \mid \{SelectCircle, SelectSquare, SelectTriangle\} \mid ShapeSystem$

where the charts named before and after the " | " are composed together and the set of signals between them is the feedback set they use for communication. $ShapeUI \mid \{SelectCircle, SelectSquare, SelectTriangle\} \mid ShapeSystem$ models the dynamic behaviour of the UI as well as the interaction between

Figure 5.5: Composed Chart for Shape UI and System

UI and system, which occurs due to the communication on the feedback signals {*SelectCircle*, *SelectSquare*, *SelectTriangle*}. The UI chart begins in state

OpenWin and the system chart begins in state *Init*, from there the dynamic behaviour of the UI to any of the states *CircleWin, SquareWin* or *TriangleWin* can be triggered by input of *I_Behaviours* and *S_Behaviours*, and each of these transitions in turn outputs a signal which triggers the corresponding transition within the system chart. The implicit interface of this chart, however, also allows the environment to provide signals such as *SelectCircle*, *SelectSquare* and *SelectTriangle* directly which can lead to a transition occurring in the system chart which has not been triggered by the UI. If we now add the chart interfaces to $ShapeUI \mid \{SelectCircle, SelectSquare, SelectTriangle\} \mid ShapeSystem$ we can prevent this from happening.

In figure 5.6 we have all of the same transitions and signals present, but we have added explicit input and output interfaces. If any signals not in the input interface (which is the set of signals listed in the box to the left of the chart) come from the environment they will be ignored. Similarly any output signals from transitions which are not in the output interface (the set of signals in the box to the right of the chart) will not be seen by the environment. Notice that the defined interfaces are for the composed chart (in fact each of the sequential charts also has an input and output interface but as we have not specified these they are assumed to be the natural interfaces of the charts), communication *between* the two sequential charts on *any* signals is therefore unaffected, it is only anything external to this composition (in this example just the environment) which is affected. Generally the output interface of our $(UI \parallel Sys)$ composed charts will be empty as we have no requirement to

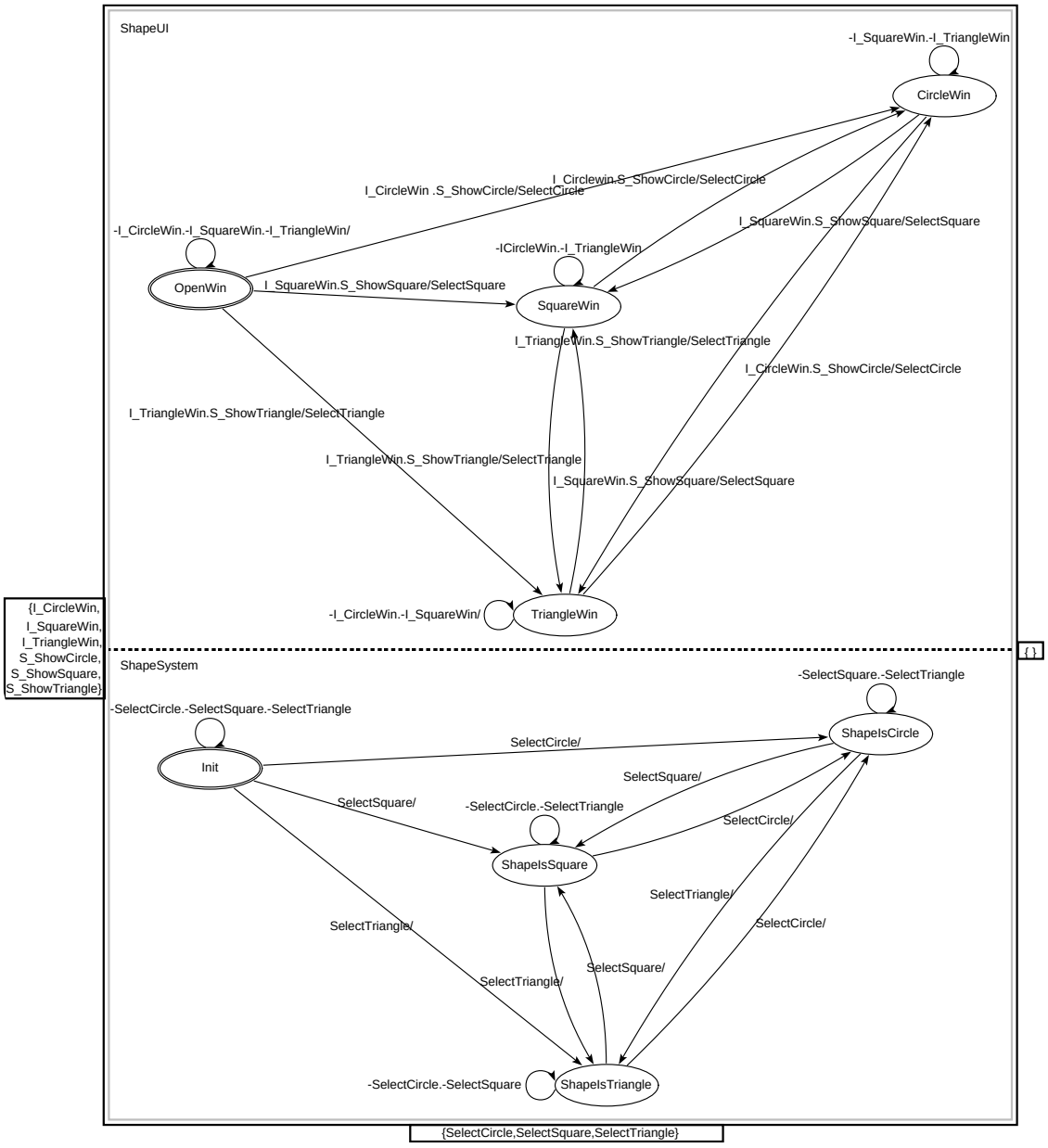


Figure 5.6: Composed IO Chart for Shape UI and System

provide signals back to the environment, but this is not a requirement of our charts, or of charts in general.

Note that the modularity of μCharts means that we do not have to describe the entire system (or UI) in a single chart, but can use both decomposition and

further composition to structure the model. This means that we can model parts of the system which have no relationship to the UI as separate charts and subsequently compose them with the rest of the model, relying on the logic of μCharts which ensures the correct meanings are preserved.

5.3.3 Theorem Proving to Ensure μchart Correctness

The μchart we have given in figure 5.6 is an example of how we expect to correctly model ($UI \parallel Sys$). In order to be sure that the semantics of this chart are as we have described we perform one more step which is to use the Z/EVES theorem prover to check that the properties we require of the chart are correct.

Using the ZooM translation tool we can derive the Z description of the system given in the μchart (the full translation is given in Appendix E). We then perform tests using Z/EVES to ensure certain properties hold. For example, we want to be certain that the interfaces act in the required manner to filter input signals of the system chart (which reflects the desire to prevent users from interacting with the system directly). The restricted composition of the chart is given in the Z section called *ShapeUISystemR* so we can submit a conjecture to Z/EVES which contains inputs which should not be seen (as they are not in the input interface), which gives us inputs to Z/EVES, and results, as follows:

```
try ShapeUISys[\inShapeR? := \{SSelectTriangle\}]


Proving gives ...

false


try ShapeUISys [\inShapeR? := \{SSelectCircle\}];


Proving gives ...

false


try ShapeUISys [\inShapeR? := \{SSelectSquare\}];


Proving gives ...

false
```

The signals *S_SelectTriangle*, *S_SelectCircle* and *S_SelectTriangle* are not possible inputs to the composition, which explains the Z/EVES result of *false*. While it is possible for *any* signal to be input to a chart in general, where we have a defined input interface the inner composition can never see signals which are not in the interface as they are filtered out of the input set. In this way the charts behave as we expect, as demonstrated by the Z/EVES result.

We can also ensure that the behaviour of the composition reflects the relation described in the *PMR*, that is we expect that given an input which is an *S_Behaviour* causing a transition in the UI chart, the system chart

should receive an input representing the related system operation and perform a corresponding transition. So, for our example chart if *ShapeUI* is in the start state *OpenWin* and *ShapeSystem* is in its start state *Init* and the signals *I_TriangleWin* and *S_ShowTriangle* are input which causes the UI chart to make the transition to the state *TriangleWin* then we expect that the System chart will receive the signal *SelectTriangle* (which is output from the UI transition and passed to the System chart via the feedback mechanism) and make the transition to the state *ShapeIsTriangle*. We give this conjecture to Z/EVES with the following input and result:

```
 try ShapeUISys [\cShapeUI := \ShapeUIOpenWin, \cShapeSystem :=
\ShapeSystemInit, \inShapeR? := \{SSShowTriangle, SITriangleWin\},
\cShapeSystem' := \ShapeSystemShapeIsTriangle,\cShapeUI' :=
\ShapeUITriangleWin, \outShapeR! := \{\} ];

instantiate (active \_) == \{ShapeUISystemR, ShapeUIShapeSystem,
ShapeUI, ShapeSystem\};

Proving gives ...
true
```

We can similarly perform this test for each of the operations in the feedback set and be satisfied that the chart behaves as required. In this way we validate the μchart model to ensure it is correct with respect to our requirements.

5.4 Validity Conditions and μCharts

The final step in modelling the (UI || Sys) as a composed μchart is to consider the validity requirements we outlined in section 5.2 and show how we can describe these as conditions upon the μchart itself to ensure that each of our charts describes a valid application. First we describe how a valid (UI || Sys) is correctly modelled using μCharts to ensure that the overall behaviour captured by the validity conditions are also true of the μchart. The PMR between a presentation model and system specification consists of related pairs of behaviours and operations describing interaction, these must be represented within the μchart as transitions with the following requirements:

R1 Each $S\_Behaviour$/Operation pair in the PMR appears as the guard ($S\_Behaviour$)/action(Operation) of at least one transition in the sequential UI μchart;

R2 Each $I\_Behaviour$ appears as the guard on at least one transition in the sequential UI μchart;

R3 All operations related to $S\_Behaviours$ appear in the feedback set of the composition.

We have already discussed the use of μChart interfaces and next we describe the requirements on these to ensure that the correct interaction and communication takes place between user, UI and system:

R4 Both sequential charts in the composition must have their natural inter-

faces.

R5 The input interface to the composition is exactly the same as the natural interface of the sequential UI μchart

These requirements are satisfied in the μchart we presented in figure 5.6, and we are similarly satisfied that the application described by that chart is valid. We will show in the next chapter of this thesis how these requirements, which relate to communication between charts, and relations between the sequential components have an interesting congruence with refinement requirements for the μCharts language.

5.5 Discussion

Our intention was to find a way to bring together the system specification and UI models into a single model which would describe the composition of system and UI. We have done so using μCharts which provides a method of joining together charts into a communicating composition. We had already developed a μchart representation of a PIM, which is a suitable representation of the UI in this context as it has all of the behaviours of the UI (and it is the behaviours that are of interest). For the underlying system we develop a μchart of the relevant parts from the specification, that is, the parts which are related to UI behaviours. For the parts of the system not related to the UI we can either model them as separate μcharts, relying on the modular nature of the μCharts language to enable us to subsequently create a complete model if required, or,

if our specification is in Z, we can use the Z semantics of the charts to provide a complete specification.

We have built on our original description of consistency between UI and system designs, which we expressed via the *PMR*, and developed definitions of interactivity and mutual interactivity between a UI design and system description. This provides the basis for a definition of validity for an application which consists of a $(UI \parallel Sys)$ which is considered in terms of the interactivity between the two models. We have also shown how this is expressed within a composed μchart using input and output interfaces, as well as the feedback mechanism, to control how each part (user, UI and system) interacts with the other.

What we now have is a single model of communicating UI and system with requirements on how interaction occurs based on the original prototypes of the UI and the specification of the system. Again we are using the designs as the basis for the formal model rather than the other way around, and as such we retain the benefits of the UCD process which have led to the development of the prototypes and UI designs. In addition, because our validity requirements include the original consistency requirements we can also use the composed model to ensure that consistency is retained if changes are made to the UI designs (we can recreate the PIM of the new design and compose this with the system chart and then check the composition is valid).

We have started to consider some of the implications of moving toward an implementation of our modelled system. In particular, our choice of seman-

tics for the μCharts language means that we must explicitly describe how to prevent erroneous behaviour within the model in order that we can be sure this is considered when implementation occurs. The final step in our software development process is to make the transformation to an implemented system. Having a single model for $(UI \parallel Sys)$ means we can now consider whether there is a refinement process we can use which is suitable for both the UI and the system, and which therefore enables us to make the transformation either in tandem (both parts of the model together) or, as is more likely, refine each part separately but retain the guarantees of correctness when we recompose the implemented parts based on the common refinement method. In our next chapter we will consider how we can begin to consider such a refinement and how the refinement theory of μCharts provides one way of satisfying our needs.

Chapter 6

Refinement

6.1 Introduction

In our work so far we have shown how we can create a link between formal specifications and informal design artefacts, using presentation models and PIMs. We have then used these models as the basis for developing a composed model (in a common formalism) of both system and UI. These steps have provided a way of linking separate development strands for systems and UIs (by way of the formal models and *PMR*) as well as the ability to consider system and UI together using μCharts. In addition we have defined validity for such compositions to ensure properties of mutual interaction between the system and the UI as well as properties of user interaction.

Our final step is to show how we can transform such a model into an implementation with the assurance that we preserve all of the properties we have described so far. That is, we want to be certain that the consistency

between UI and system (and interactivity between the two) is preserved. In addition, we would like to ensure that any properties about the UI itself, in terms of usability and satisfying user requirements, are similarly preserved. Our motivation for considering refinement is, therefore, to ensure that we can derive an implementation for our valid applications and be certain they are correct, as well as providing a means to consider the transformation of UI designs into implemented UIs with the same rigour provided by traditional methods of refinement for systems.

In order to develop such a refinement theory we first need to consider what we mean by refinement for UIs. Working in a UCD process already entails a form of refinement in that we get feedback from users which leads to changes in the designs. Similarly we need to consider the nature of the design artefacts and how this may affect the way the UI is implemented. For example prototypes may initially be created on paper, and then subsequently transferred to computer as horizontal prototypes (prototypes with partial functionality) with functionality incrementally added. The computer-based prototype may be developed in the final implementation language of the system, or may be a temporary design created using tools such as Visual Studio which allow a 'drag and drop' approach (where toolboxes of available widgets can be used to position elements with behavioural code added later as required) with as much, or little, functionality added as required. These may then subsequently be re-implemented in the system's target language. Each of these steps is a form of refinement and so we must consider how we can capture these within

a theory of UI refinement and which parts we can usefully formalise.

We will also identify the different properties of UIs which need to be considered for refinement purposes. There are three areas to take into account when we implement UIs: the visual appearance; the behaviour; and the interactivity (between UI and system as well as UI and user). Each of these may have its own requirements for refinement and we will discuss what these are and how we can use them to build a general concept of UI refinement and subsequently formalise this.

As we have discussed in chapter 1 of this thesis, the transformation from design to implementation is rarely undertaken in one step. It is more usual to move incrementally towards an implementation via stepwise refinement. This is reflected by the process we have described above for UIs where designers make small changes based on user feedback, or changes between fidelity of prototypes, and as such we expect that our UI refinement theory will similarly have a stepwise approach. We will also want to use refinement to compare UIs to consider whether they are in some sense the same, just as we do with systems. Our aim then is a refinement theory for UIs which provides everything we expect for refinement of systems in general.

To begin we will consider some general notions for refinement which underpin the formal refinement theories which exist for many of the formal languages and notations we have discussed. This overview of common refinement considerations will then be used to informally consider what UI refinement is. We then expand this to consider a formal refinement theory for UIs and discuss

the implications of this.

6.2 General Notions of Refinement

Not only are there different refinement methods for different languages, but in some cases languages may have more than one refinement theory. For example, refinement for CSP [49] may be based on either traces or failures. We may take a data refinement approach, where abstract data types are transformed into implementable data structures, or an operational approach which converts specified operations into implementable programs (or a combination of both). We may also consider strengthening post-conditions and/or weakening pre-conditions in the manner of Dijkstra [32] and Morgan [65] and reducing nondeterminism via strengthening of post-conditions.

Before we can determine a suitable approach to refinement for our UI models (and $(UI \parallel Sys)$ compositions) we first need to understand what it is for a UI to be refined and how this fits into possible approaches to refinement. We will next examine three general principles that underpin the various refinement techniques, that is, discuss the general characteristics of refinement. We will then use these as the basis for considering UI refinement and outline what are the important principles of such refinement before moving on to consider formalising these.

6.2.1 Substitutivity and Programs as Contracts

One general description we can give of refinement is the principle of substitutivity. This states that it is acceptable to replace one program with another provided it is impossible for a user to observe that the substitution has taken place. If a program can be acceptably substituted by another, then the second program is said to be a refinement of the first.

What this means is that if a user can perform a certain sequence of actions $\langle a_0, a_1, \ldots a_n \rangle$ provided by a program P_0 and we replace it with another program P_1, which allows the user to perform the same series of actions, and this elicits the same behaviour (such that the user cannot tell they are not still using P_0) then we would say that P_1 is a refinement of P_0. It is possible that P_1 provides additional actions which were not available in P_0, but when carrying out their sequence of actions the user cannot tell. By changing the programme we have not reduced the actions available to the user.

Usually when we talk about substitution in this way we are considering behaviours of systems in terms of either input/output traces, or perhaps interaction with other processes. That is, behaviour devoid of any notion of a UI or visual appearance, or even cognitive awareness of a user which would lead them to identify the substitution. For UIs, however, visual appearance and user's cognition cannot be ignored, if we substitute one UI for another and it is in any way different visually, then we expect that a user will be aware of the substitution. For the purposes of refining visual elements of the UI we cannot,

therefore, rely on this general principle. Rather than considering substitutivity, we can instead consider a closely related principle, that of considering programs as contracts.

In [65] 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 the same things they could previously (it must meet the contract) and perhaps more. So we can perform a substitution of one program by another, and it does not matter if the user is aware of the substitution, as long as we continue to meet the contractual obligations. We will refer to this as maintaining contractual utility. This approach has been considered for interactive systems in the work of Back *et al.* [7] where computation and interactions between components (including human interaction) is considered in terms of contract statements.

6.2.2 Decreasing Level of Abstraction

Another general principle of refinement is that of becoming less abstract by, for example, adding more information or becoming more precise about how data is stored or how operations are carried out. This enables us to use refinement as a means of moving from an abstract system to a concrete one in a structured manner. This must be done in a way which avoids inconsistency,

that is, as we become more precise about certain details we must preserve what we have previously decided. Each new version of the system is, therefore, a specialisation of the previous, more abstract one such that information change must be monotonically increasing (or at least non-decreasing).

For UI designs there are two areas in which abstraction can be decreased. Firstly, the design itself may lack visual detail, and so can become less abstract by defining the appearance (for example by determining things like detailed positioning of widgets, colour schemes, font usage on text labels *etc.*) Secondly, we can add more information by defining the categories of the widgets used more precisely. The formal models we have presented use a widget category hierarchy to abstractly describe widgets in terms of general behaviour (these are the hierarchy trees given in Appendix B).

In fact, the hierarchy trees themselves present a form of refinement for widgets by defining the specialisation we have referred to above. By moving down the tree the children of each node are specialisations of the parent and as such the hierarchy trees can be used both to guide widget refinement (by showing the possible choices for refinement) as well as testing correctness of refinement (by ensuring that implemented widgets are children of the abstract widgets). There is a similarity in this approach to that taken by Eisenstein and Puerta in [38] where decision trees are used to try and improve automated widget selection. Although our hierarchy trees are intended to be used in a human, rather than automated, process it would be possible to use them within an automated approach in a similar manner.

It is worth pointing out that although the hierarchy trees as presented seem to suggest that UIs are primarily WIMP-based this is not the intention. At the most abstract level we consider widgets as controls which either cause actions or respond to actions. It is possible to extend the trees downwards from these points to encapsulate different interaction possibilities by instantiating leaf nodes or adding branches with appropriate descriptions. The hierarchy itself is, therefore, generic enough to be suitable for any interaction or interface type.

6.2.3 Removal of Nondeterminism

The third principle of refinement we consider here is that of removing nondeterminism. An acceptable change to a program is one where nondeterministic behaviour is reduced or removed. If we have some program Q_0 which upon invoking action a sometimes exhibits behaviour b_0 and sometimes behaviour b_1 and the user finds Q_0 acceptable and so does not care which of these behaviours a causes, then replacing programme Q_0 with program Q_1 which always exhibits behaviour b_0 when action a is invoked will be acceptable to that user. In fact the replacement will not be detectable (so again we have the notion of substitutivity) as in order to be certain the nondeterminism had been removed the user would need to run the program an infinite number of times to be sure that behaviour b_1 never occurs. Q_1 therefore is an acceptable refinement of Q_0.

This sort of refinement is useful when using abstract models or specifications as a basis for moving to a final implementation as it allows us to ignore

details which are not important at a particular point in time. For example, in an early specification what happens when action a is invoked may be of no concern to the user so we allow the model to do anything. Later as we become more precise and we do care what happens when a is invoked we refine away the ambiguous behaviour.

6.3 Refinement for UIs

Having considered the underlying principles of refinement in general we now show how these can be applied to UI refinement. In order to do this we must consider how each of the following is affected by refinement: system functionality; UI functionality; interaction possibilities. In our descriptions above we have talked about *users* of systems, for UIs we need to be more precise and differentiate between the two different types of users. That is, there is a human user who interacts with the UI, but there is also the underlying system which interacts with the UI. When we think about contractual utility for example, we must consider both the human user and the system as users with whom there is a contract.

In order to maintain contractual utility a refined UI needs to provide at least the functionality of the previous UI (and any new functionality must be consistent with the old). Within our UI models we define functionality by way of behaviours and therefore we can use these as a basis to determine this.

The *S_Behaviours* in the models are those behaviours which allow a user

to interact with the underlying system, they are, therefore part of the contract between user and UI (these are the ways we guarantee to allow the user to interact with the system) but they are also part of the contract between UI and system (these are the ways we expect the UI and system to interact with each other). To maintain contractual utility with the user the new UI must provide all of the *S_Behaviours* as the previous version, but we could in theory include more *S_Behaviours*. However, to maintain contractual utility with the system we must provide exactly the same *S_Behaviours* as the previous version, that is, we cannot include new behaviours.

It may appear that this requirement on *S_Behaviours* is unnecessarily strict. What if a UI provides behaviours a, b and c to a user, and a replacement provides a, b, c and d? This would seem to be a suitable refinement based on our previous description of contractual utility. However, in terms of the contract between UI and system, allowing the UI to increase *S_Behaviours* in this way means that the UI offers interaction possibilities to the user which may not be defined in the underlying system, that is, there is no guarantee that these new *S_Behaviours* will be supported by the system. If we add a widget to our new UI which provides behaviour d to a user, it may be that the system is implemented without such a behaviour. While the user is no worse off as far as the behaviour of the application is concerned (in that they were unable to perform function d previously) we must also consider the usability aspect of refinement (we will discuss this in more detail shortly). Providing a widget on a UI which does not do what the user expects will reduce the

usability of the UI and therefore we might expect that the user would not be happy with the substitution.

We can use the *S_Behaviour* equivalence defined in section 4.3.4 to determine whether this property of *S_Behaviours* is being met. For two arbitrary UIs, UI_A and UI_C we state that our first requirement for contractual utility is:

$UI_A \equiv_{SBeh} UI_C$

The *I_Behaviours* of the UI models are not related to the underlying system behaviour, and as such we only need consider the human user when we look at the possibilities for these. Once again the user will expect to have all of the same UI behaviours in the new UI as they did with the old, but we can also provide additional *I_Behaviours* (as we do not have the same risk of these being unsupported by the system functionality). We therefore state that for arbitrary UIs UI_A and UI_C our second requirement for contractual utility is:

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

Where $I\_Beh[P]$ is the syntactic function that returns the identifiers for all of the *I_Behaviours* in P (defined in the same manner as $B[P]$). The definition of contractual utility is, therefore, given by these requirements on *S_Behaviours* and *I_Behaviours*.

Allowing the addition of *I_Behaviours* in this way is different from the sort of extension of functionality allowed by retrenchment and described in work such as [8] for example. We do not permit arbitrary new system behaviour, but rather the addition of *I_Behaviours* provides new ways to access already specified behaviour. As such it may be considered a type of removal of nonde-

terminism in that we make specific how a user will move around the UI where we may previously have abstracted such concerns.

As an example consider the three UIs for the Shape Application given in figure 6.1. The definition sections of the presentation models for each of these designs are as follows:

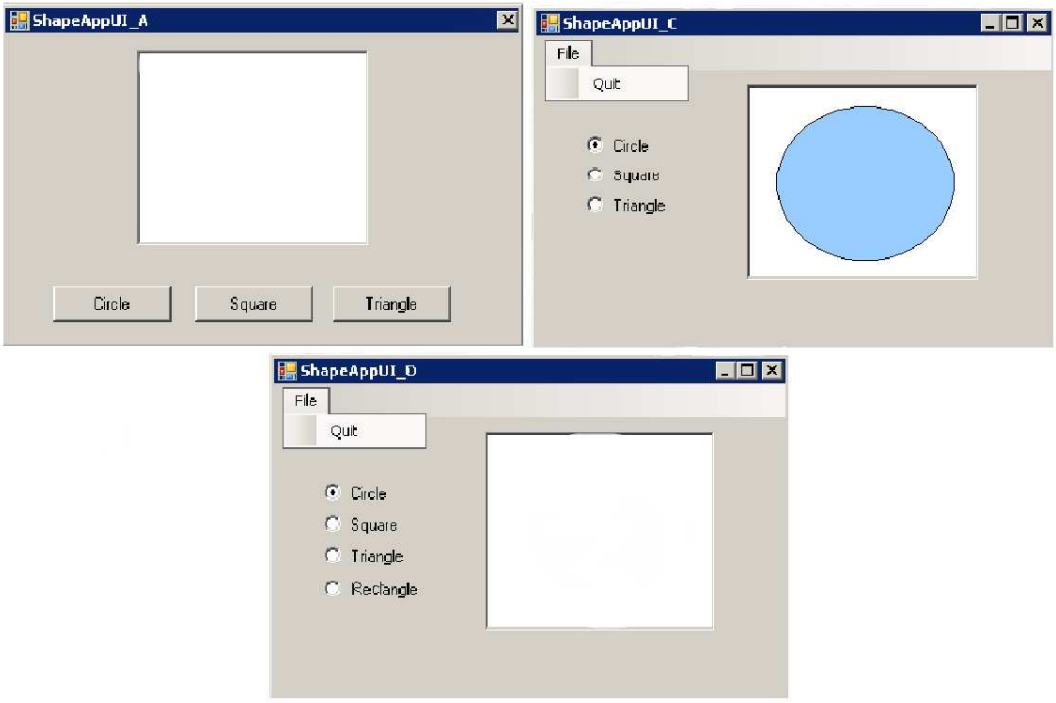


Figure 6.1: Three UIs for Shape Application

UI_A *is* $(QuitButton, ActCtrl, (Quit)),$

$(CButton, ActCtrl, (S\_ShowCircle)),$

$(SButton, ActCtrl, (S\_ShowSquare)),$

$(TButton, ActCtrl, (S\_ShowTriangle)),$

$(SFrame, Responder, (S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle))$

UI_C *is* $(QuitButton, ActCtrl, (Quit))$,

$(CRButton, Button, (S\_ShowCircle))$,

$(SRButton, Button, (S\_ShowSquare))$,

$(TRButton, Button, (S\_ShowTriangle))$,

$(SFrame, Responder, (S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle))$,

$(FileMenu, Container, ())$,

$(QuitItem, ActCtrl, (Quit))$,

$(MinButton, ActCtrl, (I\_MinWin))$,

$(MaxButton, ActCtrl, (I\_MaxWin))$

UI_D *is* $(QuitButton, ActCtrl, (Quit))$,

$(CRButton, Button, (S\_ShowCircle))$,

$(SRButton, Button, (S\_ShowSquare))$,

$(TRButton, Button, (S\_ShowTriangle))$,

$(RRButton, Button, (S\_ShowRectangle))$,

$(SFrame, Responder, (S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle,$

$S\_ShowRectangle))$,

$(FileMenu, Container, ())$,

$(QuitItem, ActCtrl, (Quit))$,

$(MinButton, ActCtrl, (I\_MinWin))$,

$(MaxButton, ActCtrl, (I\_MaxWin))$

Comparing the respective sets of *S_Behaviours* and *I_Behaviours* of UI_A and UI_C, which are:

$$S\_Beh[UI\_A] = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$$

$$I\_Beh[UI\_A] = \{\}$$

$$S\_Beh[UI\_C] = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$$

$$I\_Beh[UI\_C] = \{I\_MinWin, I\_MaxWin\}$$

we find that the following properties are true:

$$UI\_A \equiv_{SBeh} UI\_C$$

$$I\_Beh[UI\_A] \subseteq I\_Beh[UI\_C]$$

Based on our earlier premise we would, therefore, state that the contractual utility of UI_A is maintained by UI_C. We do not say that UI_C refines UI_A as we will show shortly that there are other considerations beyond that of contractual utility.

If we compare the *S_Behaviours* and *I_Behaviours* of UI_A and UI_D, which are:

$$S\_Beh[UI\_A] = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$$

$$I\_Beh[UI\_A] = \{\}$$

$$S\_Beh[UI\_D] = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle,$$
$$S\_ShowRectangle\}$$

$$I\_Beh[UI\_D] = \{I\_MinWin, I\_MaxWin\}$$

then contractual utility is not maintained, because $UI\_A \not\equiv_{SBeh} UI\_D$, and, therefore, UI_D is not a refinement of UI_A. If we implement UI_D we have no guarantee that the underlying system will provide the *S_ShowRectangle* behaviour and therefore the interaction between UI and System may fail.

Our next consideration for refinement is that of reducing abstraction, or becoming more detailed. We have already discussed how the widget hierarchy trees can be used to check this property and also act as a guide to assist with correct widget selection. Defining widgets more precisely is one way of becoming less abstract, but we may also have a refinement where the widget remains unchanged (previous versions of the UI may have already precisely defined the widgets to be used). If a widget category *has* changed then we need to make sure we have not become more abstract (*i.e.* selected a widget from higher up the hierarchy tree) and also that we have chosen a widget which is either a descendant of the previous one, or, if the previous widget was a leaf node in the hierarchy tree (*i.e.* an actual widget rather than an abstract description in terms of behaviour) then the replacement widget must have the same parent (so we may change our mind about the actual widget we want to

use and replace it with something at the same level of abstraction).

In our examples given in figure 6.1 UI_A has three widgets which are described as *ActionControls* for the behaviours *S_ShowCircle*, *S_ShowSquare* and *S_ShowTriangle* and both UI_C and UI_D have *Buttons*. The widgets have become more precise and we can check the choice is correct by ensuring that *Buttons* are descendants of *ActionControls*. If we were to produce a design which used a *Slider* to control the choice of shape then we would describe this as an incorrect refinement as a *Slider* is not a descendant of *ActionControl*. This is an example of how the widget hierarchy supports design guidelines by avoiding inappropriate use of widgets. For example, the GNOME Human Interface Guidelines [41] 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 widget hierarchy trees to support refinement enables designers to avoid such incorrect usage without the need to refer to the guidelines, and additionally may support more inexperienced designers in this area.

Defining the appearance of the UI relates to visual aspects such as position and style of widgets, as well as the overall layout appearance (including things such as background colours, font choice *etc.*) Some of these aspects may not be fixed within the application itself, but set during the application's runtime by the local operating system. For example the Java programming language uses

a *look-and-feel* concept where the choice of style of the widgets is controlled by the window management system of the underlying operating system. The low-level details of UI appearance are not included in the formal models of UI designs and so we cannot use these to determine whether such changes have been made. We rely on the visual appearance to inform us of visual changes and as such detail does not affect behavioural properties of the UI we will not consider them as part of refinement.

So far we have only considered behaviour properties of UIs for refinement, but another important consideration is that of usability. It is not enough to ensure we maintain the required behaviour when we refine a UI, we must also ensure that changes we make do not adversely affect the user by making the system harder to use. In terms of contractual utility we must at least maintain the usability of the previous version of the UI. We have shown in chapter 4 how presentation models and PIMs can be used to consider some aspects of usability, and it is these aspects that we can similarly consider during refinement. So, for example we would want to ensure that we do not introduce deadlock into the UI where it was not present previously, also the new UI should have the same reachability properties as the original, and we should also ensure that consistency is maintained. As we can add new $I\_Behaviours$ when we refine a UI we should also check that we do not introduce excessive chaining (as described in chapter 4) and that the UI retains its responsiveness.

Consider the design give in figure 6.2 which is another multi-window design for the shape application. The PIMs for this design, along with the PIM for

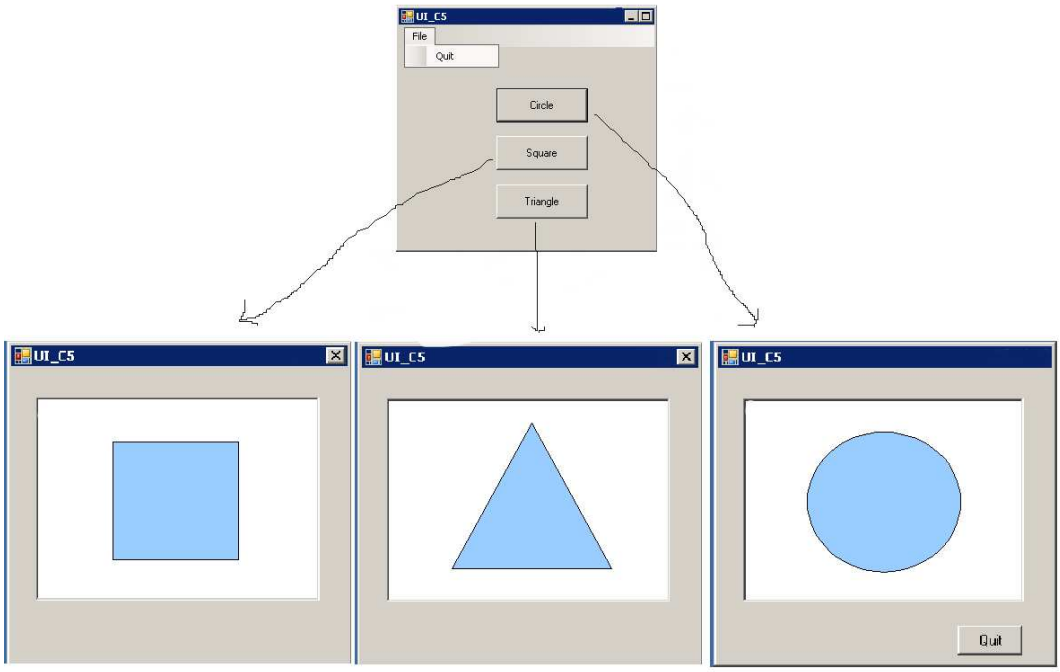


Figure 6.2: Multi-Window Shape Application

UI_A from figure 6.1, are given in figure 6.3. The PIM for *UI_A* consists of a

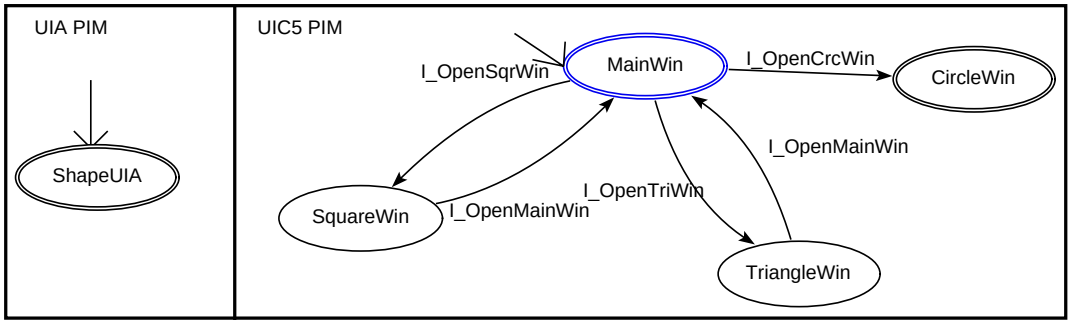


Figure 6.3: PIMs for UI_A and UI_C5

single state, and so we can immediately be satisfied that it has strong reachability and no deadlock. The PIM for *UI_C5*, however, does not have strong reachability; it is not possible to reach the *MainWin* state from the *CircleWin*

state (in fact we cannot reach any other state from *CircleWin*). We cannot therefore state that UI_C5 maintains usability as it has more restrictions on the availability of behaviours than UI_A, this breaks our requirement and so we state that UI_C5 is not a satisfactory refinement of UI_A.

To summarise, we can use standard refinement concepts such as contractual utility and the reduction of abstraction to consider refinement for UIs and UI designs. Using presentation models and PIMs we can compare behaviours of different UIs and use these to determine whether or not contractual utility has been maintained. We can also use the models to consider widget refinement as well as usability properties and ensure that these are also maintained during refinement.

Before we move on to discuss formalising refinement for UIs we discuss how the refinement we have described so far relates to more traditional methods of refinement for UIs. That is, we consider the way in which designers traditionally transform their designs into implementations, which we refer to as intuitive refinement, and show how we can both support and enhance this using the methods we have described above.

6.3.1 Intuitive Refinement

We have already noted that 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.

We can also extend this concept to not only mean the process of transforming a design, but also the ability of a designer to compare different designs and have some understanding (based on the user requirements for the system and their experience of design principles) as to whether or not one refines the other. We can compare this with the approach to UI refinement we have described in the previous section.

Consider the two UIs for the shape application given in figure 6.4, for example. The left-hand side shows a paper prototype and the right-hand side an implementation. The implementation is less abstract, has defined widgets, has a defined layout and appearance, and maintains both the usability and contractual utility of the original prototype. So, based on our informal understanding of UI refinement we state that this is a satisfactory refinement. However, the visual appearances of the two UIs are close enough that the designer may already 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. Of course, the intuition

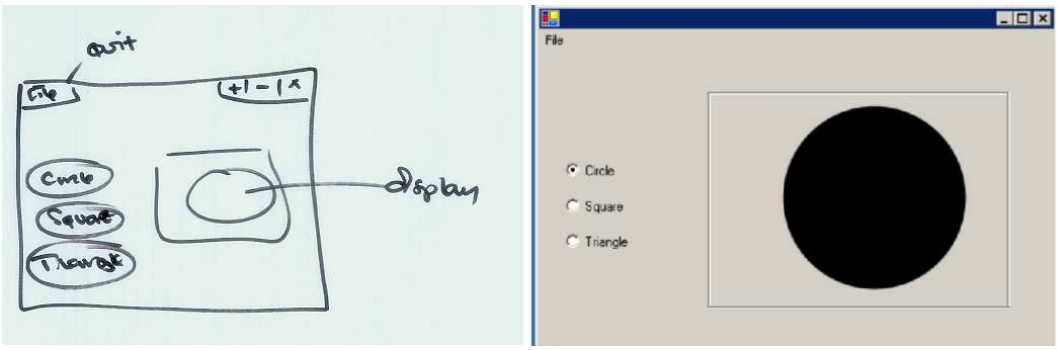


Figure 6.4: Prototype and Implementation of Shape Application

is supported by the simplicity of this application. But it is possible that this may be equally true of a more complex application and that if our design approach is incremental the gap between the final design and the implementation itself may be small enough to support the idea of intuitive refinement, that is the appearance of the two UIs may be similar enough to give some intuition that their behaviour may be the same.

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 when such changes happen towards the end of the design process, where fine-tuning of the design is taking place, they are most likely to be supported intuitively as very small, incremental changes are being made.

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.

To summarise, intuitive refinement relies on design experience and either incremental changes or visual similarity between designs. But, it provides no guarantees of correctness and may be misleading, as we will show with our next example.

One of the problems with intuitive refinement is that as applications become more complex, and particularly as numbers of different windows, dia-

logues *etc.* begins to increase, then it becomes more likely that our intuition may not be correct as it is harder to mentally process the whole system. Recall, for example, the shape application UI with multiple windows which we gave in figure 6.2.

Whilst this may appear to intuitively refine the original UI_A design, we have already shown that there is in fact a usability weakening due to a lack of total reachability which is easy for a designer to miss. This is an example of what is still a small and simple application where we might be deceived by intuitive refinement.

This suggests that while there are some parts of refinement which may be intuitively understood by designers, for anything beyond small, simple applications this is unlikely to be enough. The refinement approach we have discussed so far is able to support intuitive refinement (for example by assisting with correct widget selection) and ensure that this does not lead to mistakes such as we have described above. So, our informal notion of UI refinement is already useful in that it may support those designers refining intuitively (by giving them another means to guide the transformation).

In addition, there are cases where it is unlikely that any notion of intuitive refinement between designs will exist at all, such as UIs designed for different hardware where modes of interaction are different (we consider this further in section 7.4). However, it may also be the case when we redesign a UI without changing hardware. For example, suppose we want to make the shape application into an instructive game for children. The application should still

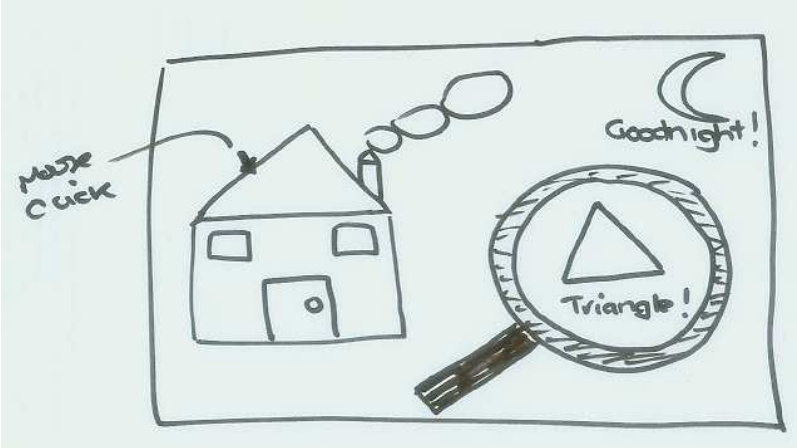


Figure 6.5: Children's Version of Shape App

provide the same functionality, that is, allow a user to display a circle, square or triangle, but the new version uses a picture-book style interface where children can look at a picture which is made up of the three shapes, and by clicking on the different parts of the picture view the shape that is currently selected. An example of a possible UI for such an application is given in figure 6.5.

It is not immediately apparent that this has the same functionality as that of the shape application UIs given in figure 6.4. If we look at the presentation models for these UIs, however, which are:

$ChildShapeApp\ is\ (MoonIcon, ActCtrl, (Quit)),$

$(GNTxt, Label, ()),$

$(MagnifyIcon, SValRes, (S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle)),$

$(HouseBody, ActCtrl, (S\_ShowSquare)),$

$(Door, ActCtrl, (S\_ShowSquare)),$

$(LWindow, ActCtrl, (S\_ShowSquare)),$

$(RWindow, ActCtrl, (S\_ShowSquare)),$

$(Chimney, ActCtrl, (S\_ShowSquare)),$

$(DoorHandle, ActCtrl, (S\_ShowCircle)),$

$(CloudOne, ActCtrl, (S\_ShowCircle)),$

$(CloudTwo, ActCtrl, (S\_ShowCircle)),$

$(CloudThree, ActCtrl, (S\_ShowCircle)),$

$(Roof, ActCtrl, (S\_ShowTriangle)),$

$(ChimneyPot, ActCtrl, (S\_ShowTriangle))$

$ShapeApp\ is\ (FileMenu, Container, ()),$

$(QuitMenuItem, ActCtrl, (Quit)),$

$(QuitIcon, ActCtrl, (Quit)),$

$(ShapeFrame, SValRes, (S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle)),$

$(CircleButton, ActCtrl, (S\_ShowCircle)),$

$(SquareButton, ActCtrl, (S\_ShowSquare)),$

$(TriangleButton, ActCtrl, (S\_ShowTriangle)),$

$(MinButton, ActCtrl, (I\_MinWindow)),$

$(MaxButton, ActCtrl, (I\_MaxWindow))$

and then consider the properties we gave previously for *I_Behaviours* and *S_Behaviours*, we can show that in fact a refinement does hold. That is:

$ChildShapeApp \equiv_{SysFunc} ShapeApp$ and $I\_Beh[ChildShapeApp]$

$\subseteq I\_Beh[ShapeApp]$

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. In general intuitive refinement relies on designer experience and either small incremental changes or a strong visual similarity between designs/implementations. 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.

It is important to remember that UI refinement is not intended to replace intuitive refinement for designers, or activities such as usability testing which traditionally goes alongside it, but rather refinement acts as a support to these activities. In the next section we look at formalising refinement which will provide a way of including UI refinement as part of the overall formal process.

6.4 Formalising Refinement Using μCharts

The refinement description for UIs based on contractual utility (which incorporates both behavioural and usability properties) already provides an enhancement to intuitive refinement and allows us to consider refinement between different designs (or implementations) as well as help guide the transformation from abstract design to concrete implementation. Our reason to now formalise this is based on the desire to have a precise and unambiguous method (*i.e.*

a formal method) which can be followed in order to guarantee correctness of refinement. It is not the intention that UI designers use the formal process when refining UIs, but rather they can rely on the informal description of contractual utility (which supports intuitive refinement) and be certain that there is an underlying theoretical basis which guarantees correctness. In addition, as we can use μCharts for system refinement we can be sure consistency is maintained throughout the separate refinement processes and the monotonicity property (which we discuss in section 6.4.2) will ensure that refinement of either system or UI will imply a correct refinement for $(UI \parallel Sys)$.

Another advantage of formalising is that it is then possible to develop tools to support our methods. Although that is not one of the aims of this research we do feel it is important for future work to provide possibilities for such tools to be developed.

In chapter 5 we showed how UIs and $(UI \parallel Sys)$ can be modelled using μCharts, and we now use this as the basis for formalising refinement. In [80], Reeve describes refinement for μcharts in terms of both traces and partial relations, and subsequently shows that they are equivalent. The semantics, and refinement theory, for μCharts are complex, and as such I introduce only as much detail of these as is necessary to understand their use within my refinement methods. The contractual utility requirements we have described for UI refinement most naturally lends itself to trace refinement. This is because traces of charts show the behaviour (both system and UI) which affects the interaction both between UI and system and UI and user. We start then by

showing how μChart's trace refinement can be used to capture our requirements for UI refinement.

The traces of a μchart are the pairs of sequences of input and output sets of signals that model the behaviour of the described system. A trace has the form (i,o) where i has the form $\langle s_o^i, s_1^i, \ldots \rangle$ and o has the form $\langle s_o^o, s_1^o, \ldots \rangle$, and input signals in set s_k^i lead to the output signals in set s_k^o. This is an abstraction of the state-based view and describes only the interactions of the charts, and as such it fits neatly with our PIM description of a UI which is a similar abstraction.

There are two distinct types of trace refinement in μCharts. The first is *behavioural* refinement which describes what are the allowable behavioural changes that can be made to a chart and the second is *interface* refinement which describes what changes to the input and output interfaces can be made. It is also possible to combine the two. The definitions given for each of these are as follows:

Definition 6 ([*80*], 5.2.1) *For arbitrary charts A and C, behavioural refinement ($\sqsupseteq_b$) is:*

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

i ranges over the sequences of sets of input signals and o over the sequences of sets of output signals and $i_{\triangleright in_x}$ restricts the range of the sequence i to the

signals in the set in_x (which is the input interface), and similarly for $o_{\triangleright out_x}$. The output interface (out_x) is extended by the addition of $\perp$ which is used here to represent chaotic output, that is, if the chart behaves chaotically we cannot say what the output signals will be and so use the symbol $\perp$ to represent this. $[\![A]\!]$ is the set of all bindings for the chart A (similarly for $[\![C]\!]$), by which we mean all possible combinations of values that each of the observations of the component schemas used to describe the chart may take. The placeholder $_x$ can be replaced by any of *dn* for do-nothing, ρ-*chaos* for partial chaos, τ-*chaos* for total chaos or *fc* for firing conditions, that is it can be instantiated to show which of the μCharts semantics are being used (from those described in section 5.3.1).

Informally the definition states that C refines the behaviour of A if and only if C's observable behaviour (by which we mean the restricted traces) is a subset of A's in a particular context (which is defined by the input and output interfaces).

Definition 7 ([*80*], 5.3.1, 5.3.5) *For arbitrary charts A and C input refinement* ($\approx_I$) *and output refinement* ($\approx_O$) *are:*

$$C \approx_{\mathrm{I}} A =_{def} \forall\, i;\ o \bullet (i_{\triangleright(in_C)}, o) \in [\![C]\!] \Leftrightarrow (i_{\triangleright(in_A)}, o) \in [\![A]\!] \wedge out_C = out_A$$

$$C \approx_{\mathrm{O}} A =_{def} \forall\, i;\ o \bullet (i, o_{\triangleright(out_C)}) \in [\![C]\!] \Leftrightarrow (i, o_{\triangleright(out_A)}) \in [\![A]\!] \wedge in_C = in_A$$

The input and output interfaces define the context, or environment, we assume for the chart (in the (UI || Sys) charts they control the ability of

the external user to interact with the system and as such give the context of use). The definition of input refinement models the fact that placing a chart in a new context (by expanding or reducing the input interface) should not change the chart's observable behaviour. This may seem surprising (given that we have new signals and a new context we might expect different observable behaviour), however the purpose of input refinement is to allow for the fact that we may have a partial specification (which ignores some signals) and we subsequently define the chart's behaviour with respect to those signals. Output interface refinement on the other hand may change the observable behaviour (by allowing more signals to be output to the environment) but it may not decrease the reactivity of the chart, so for any input sequence a refined chart must exhibit at least the same amount of output information as the original.

The definition of behavioural refinement requires that both input and output interfaces remain unchanged, so in cases where both behavioural and interface refinement occurs we have the following definition:

Definition 8 ([*80*], 5.4.1) *For arbitrary charts A and C, behavioural and interface refinement ($\sqsupseteq$) is:*

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

where $_x$ is again used as a placeholder for any of the possible semantics.

This is the same as the definition of behavioural refinement without the condition that input and output interfaces are the same for both charts. However,

behavioural and interface refinement must satisfy an additional rule, which is split into four disjoint cases based on the relationship between input and output interfaces of the charts. The four cases are ([80], Proposition 5.4.2):

Case 1: $in_A \subseteq in_C \wedge out_A \subseteq out_C$

Case 2: $in_A \subseteq in_C \wedge out_A \subset out_C$

Case 3: $in_A \subset in_C \wedge out_A \subset out_C$

Case 4: $in_A \subset in_C \wedge out_A \subseteq out_C$

The associated rules require that there are intermediary charts which are interface or behavioural refinements of A and C, that is they prevent changes to both the behaviour and the interface in one step. We will introduce these rules as necessary during our subsequent examples.

Recall that we have stated in chapter 5 that we would use the *total chaos* semantics of μCharts (and gave our reasons for doing so). Under this semantics if an input is seen when the chart is in a state where no behaviour is defined for that input, then the chart may exhibit any behaviour, that is, it behaves chaotically. In our example charts we must, therefore, ensure that we include the necessary transitions to prevent such chaotic behaviour in either one or both of the charts (the abstract and concrete) otherwise we have no guarantee that they will behave as specified. If we do not include such transitions in our abstract chart (such that it may sometimes behave chaotically) then we should refine away the chaotic behaviour in the concrete chart. This will become more obvious when we consider (UI || Sys) composed charts as our PIMs should contain no such nondeterminism (being based on presentation models which

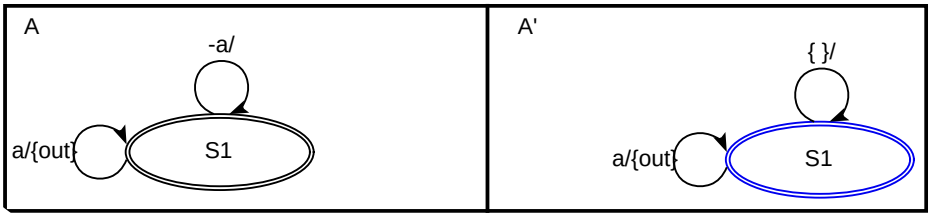


Figure 6.6: Design UI_A and UI_C

are, for the reasons given previously, deterministic) but our system models may be nondeterministic. One way to ensure deterministic behaviour is to create loop transitions for each state which have a guard which is the negation of any signal for which there is a defined transition from that state and which has no output. For example, if a chart has a state A which has defined transitions on inputs b or c then we can add a loop transition with the guard $-b.-c/$ meaning that if neither signal is present the chart should do nothing, *i.e.* remain in the same state and output nothing. As this is a common requirement for our UI charts we introduce a syntactic shorthand for input signals which is '{}' to indicate no inputs, this allows us to define behaviour when none of the described input signals are present and is useful when we consider filtering of signals via the input interface. For example the charts in figure 6.6 have defined behaviour when signal a is seen and also when signal a is not seen, chart A' uses the syntactic shorthand we have just described but has the same meaning as A (assuming a is the only possible input signal).

Returning to the shape application and example interface designs we now give an example of using trace refinement and show how this relates to con-

tractual utility. Figure 6.7 shows two different designs for the UI of the shape application, in figure 6.8 we show the sequential μcharts for each of these designs.

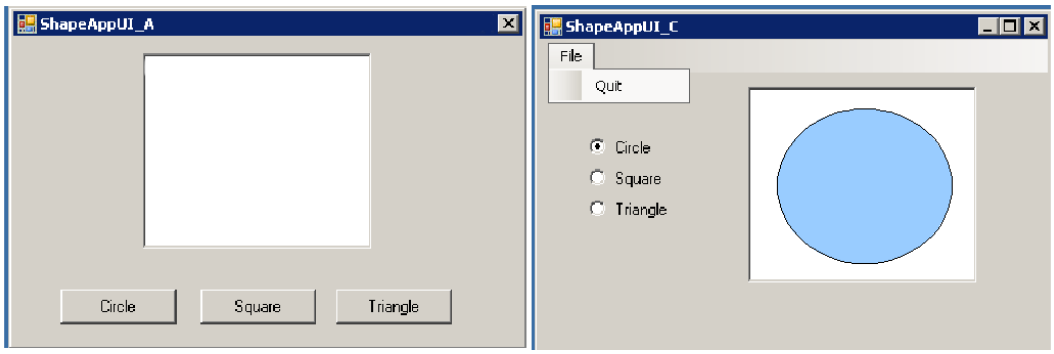


Figure 6.7: Design UI_A and UI_C

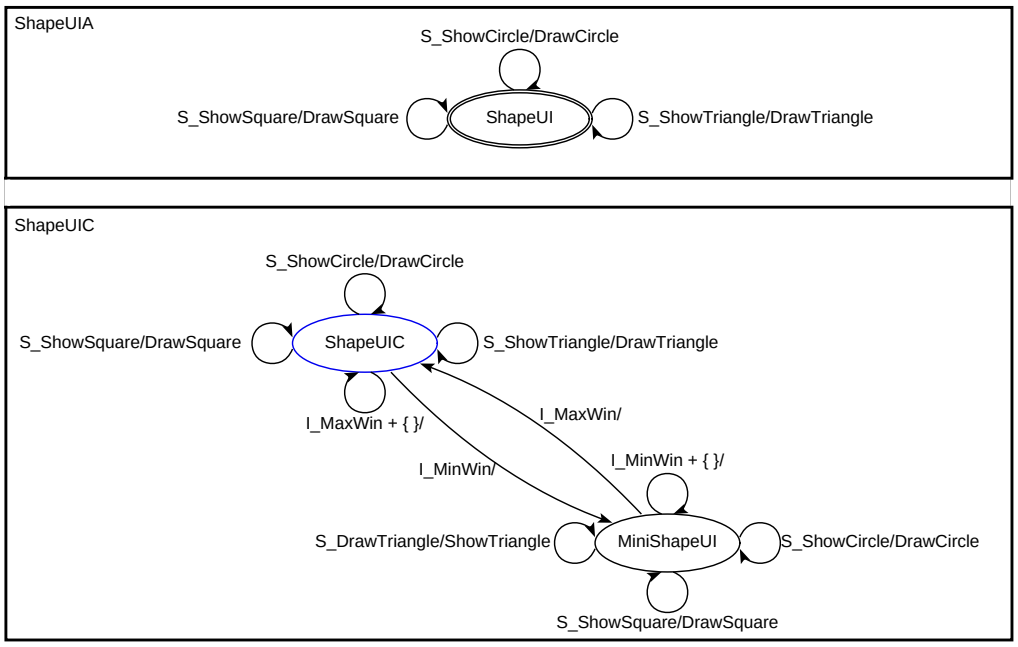


Figure 6.8: Sequential Charts for UI_A and UI_C

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:

$$in_A = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$$

$$out_A = \{DrawCircle, DrawSquare, DrawTriangle\}$$

$$in_C = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle, I\_MinWin, I\_MaxWin\}$$

$$out_C = \{DrawCircle, DrawSquare, DrawTriangle\}$$

Based on our description of contractual utility we can state that UI_C is an acceptable replacement for UI_A as it has equivalent system functionality and the UI functionality of the original design is a subset of that of the new design. Usability properties of reachability and lack of deadlock have also been preserved. We now consider the traces of the charts to see if we can likewise deduce a refinement.

The two charts have different transitions as well as different input interfaces and this is, therefore, an example of both behavioural and interface refinement. The rule required to show refinement is that of case 3 where both interfaces of A are subsets of those of C, the relevant rule for this case is ([80], Proposition 5.4.2):

$$\frac{\exists\, B;\ B' \bullet ShapeUIC \approx_O B' \wedge B' \approx_I B \wedge B \sqsupseteq_b ShapeUIA}{ShapeUIC \sqsupseteq_{\tau-chaos} ShapeUIA}$$

We must determine if charts B and B' exist such that B is a behavioural refinement of *ShapeUIA*, B' is an input interface refinement of B and *ShapeUIC* is an output interface refinement of B'. The output signals for all of the charts are the same, so trivially B' is exactly the same chart as *ShapeUIC*. The chart B is, therefore, an input interface refinement of *ShapeUIC*, and the only allowable change is to the input interface. B is therefore visually the same as *ShapeUIC* but has the following input interface:

$\{S\_ShowSquare, S\_ShowCircle, S\_ShowTriangle\}$

In order for the refinement to hold, all restricted traces of *ShapeUIC* must be restricted traces of B, and vice versa. The only possible traces of *ShapeUIC* which may be different from those in B are those involving the signals I_MaxWin and I_MinWIn as these are the signals the interfaces differ on. Any trace in *ShapeUIC* which has an input of I_MaxWin or I_MaxWin has the corresponding output of $\{\}$, and chart B, which filters out these signals, has defined behaviour for input of $\{\}$ which is to output $\{\}$ and remain in the same state. Traces on these inputs are therefore the same for both charts. The states *ShapeUIC* and *MiniShapeUI* in the charts are symmetric in terms of possible behaviours (that is, they perform the same behaviour for all inputs and give the same traces such that it is not possible to determine which state we are in from the traces alone) so the change in state does not affect overall chart behaviour (which is defined by the traces). Therefore B is a satisfactory input interface refinement of *ShapeUIC*. Now in order for B to be a behavioural refinement of *ShapeUIA* all (restricted) traces of B must be (restricted) traces

of *ShapeUIA*. The charts have the same input interfaces and the only transitions of B not in *ShapeUIA* (which may give rise to different traces) rely on signals which are not *ShapeUIA*'s interface and will therefore never be seen by the chart. The refinement therefore holds.

This example shows that is is possible to use the trace refinement of μCharts to show a refinement between UIs which has also been shown by our definition of contractual utility. Next we explain the correspondence between the two in order to show that this is true for all UI refinements and not just for the example we have given.

6.4.1 Trace Refinement and Contractual Utility

To show that trace refinement of μCharts correctly formally describes our informal description of contractual utility we consider each of our informal requirements and show how they relate to trace refinement.

Our first requirement for UI refinement was that the sets of *S_Behaviours* for both UIs should be the same. Based on the validity requirements given in section 4 we know that there are transitions in the UI μchart for each *S_Behaviour*, where the *S_Behaviour* is the input signal and the related operation (from the *PMR*) is the output signal. Traces for the original chart will,

therefore, contain the following singleton sets:

$$(\langle\{S\_Beh_0\}\rangle, \langle\{Op_0\}\rangle)$$

$$(\langle\{S\_Beh_1\}\rangle, \langle\{Op_1\}\rangle)$$

$$\vdots$$

$$\vdots$$

$$(\langle\{S\_Beh_n\}\rangle, \langle\{Op_n\}\rangle)$$

where the *S_Behaviours* range from $S\_Behaviour_0$ to $S\_Behaviour_n$.

Removing any of these transitions as we update the design (which would break contractual utility) will likewise cause trace refinement to fail as it will introduce nondeterminism into the chart. Consider the two charts in figure 6.9. The UI represented by chart C does not maintain the contractual utility of

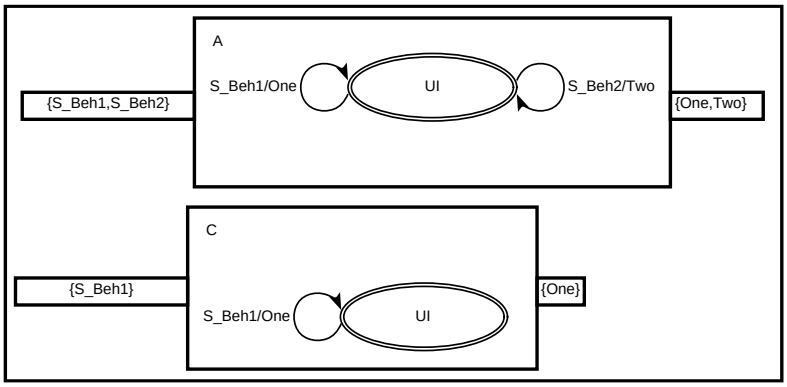


Figure 6.9: Breaking Contractual Utility

chart A as it has fewer *S_Behaviours*. When we look at the possible traces for the charts we see that removing the transition with one of the *S_Behaviours* leads to nondeterminism in chart C. The signal is not in the input interface and

so is filtered out leading to input of {} for which there is no defined behaviour and so the chart behaves chaotically. Even if we had added a loop transition with the guard {}/{} we still get a different trace on input of *S_Beh*2 as the original chart will output *Two* and the new chart outputs nothing.

Our validity conditions require that all *S_Behaviours* are represented as guards on transitions which have the related operation as output. It is, therefore, always the case that breaking contractual utility by removing one of these transitions will similarly lead to chaotic behaviour or new traces in the new chart for the reasons shown, so that trace refinement will not hold. Rather than removing one of the *S_Behaviours* another way to break contractual utility is to introduce a new behaviour (as we did earlier in this chapter when we tried to add the behaviour *S_ShowRectangle* to the shape application). Again this leads to the introduction of a trace to the chart of the new UI which is not seen in the original. Consider the two charts in figure 6.10. The traces of

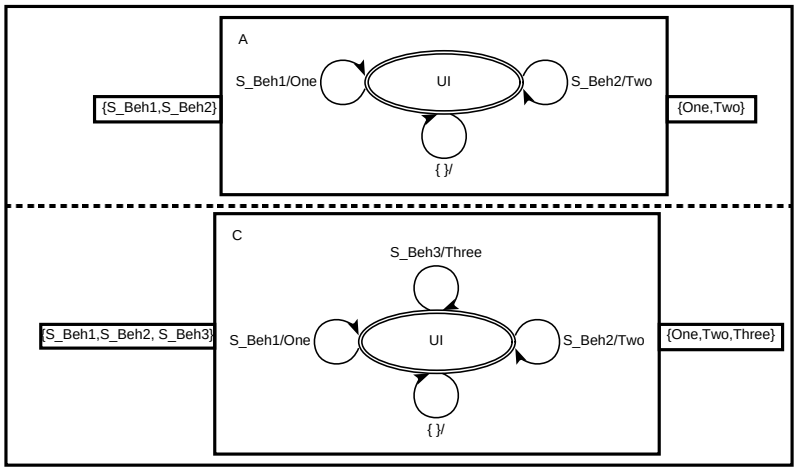


Figure 6.10: Breaking Contractual Utility

chart C contains the singleton trace:

$$(\langle\{S\_Beh3\}\rangle, \langle\{Three\}\rangle)$$

whereas chart A will filter out the signal S_Beh3 (as it is not in its input interface), which gives the trace:

$$(\langle\{S\_Beh3\}\rangle, \langle\{\}\rangle)$$

and so trace refinement does not hold. It is always the case that introducing a new *S_Behaviour* transition in this way will have this effect, as the signal will not exist in the input interface of the original chart.

The property required of *I_Behaviours* for contractual utility is that the set of *I_Behaviours* of the original UI must be a subset of, or equal to, the set of *I_Behaviours* of the new UI. We cannot, therefore, remove any of the original *I_Behaviours*. Each *I_Behaviour* is represented in the μchart as a transition between states which has no output, such that traces of the chart contain the following singleton sets:

$$(\langle\{I\_Beh_0\}\rangle, \langle\{\}\rangle)$$
$$(\langle\{I\_Beh_1\}\rangle, \langle\{\}\rangle)$$
$$\vdots$$
$$(\langle\{I\_Beh_n\}\rangle, \langle\{\}\rangle)$$

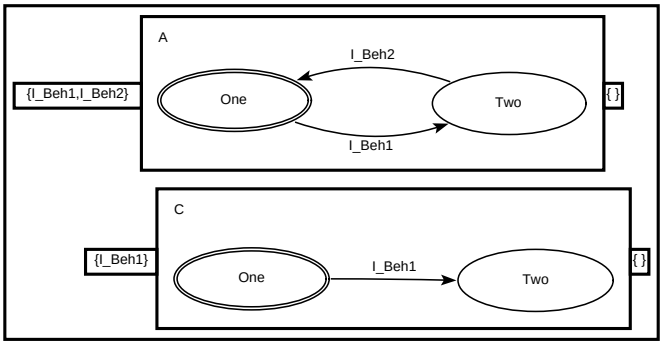


Figure 6.11: Breaking Contractual Utility

where the $I\_Behaviours$ range from I_Beh_0 to I_Beh_n.

Removing one of the $I\_Behaviour_i$ breaks contractual utility, and if we examine the respective charts for such UIs we can show that trace refinement will likewise fail. Consider the charts in figure 6.11. In order to show a refinement we must satisfy one of the conditions for interface and behavioural refinement, which in this case is:

Case 4: $in_A \subset in_C \wedge out_A \subseteq out_C$

This requires there to be suitable intermediary charts such that:

$$\frac{\exists\, B;\ B' \bullet C \approx_I B' \wedge B' \sqsupseteq_b B \wedge B \approx_O A}{C \sqsupseteq_{\tau - chaos} A}$$

We make no changes to the output interface (this is because transitions involving $I\_Behaviours$ have no output) and so intermediary chart B is trivially the same as chart A. Chart B' is an input interface of chart C and as such the only difference is that it has input interface $\{I\_Beh1, I\_Beh2\}$. However, by allowing the chart to accept signal I_Beh2 we introduce nondeterminism as

no behaviour is defined for this signal and so the chart can behave chaotically. It is always the case that removing all transitions for one of the *I_Behaviours* will have this effect.

We have looked at possible changes to UIs which lead to charts which do not refine each other, but have not discussed interface refinement which may also lead to contractual utility being broken. However, the refinement of the UI is (we expect) being undertaken by the UI designers and as such they will be relying on our informal notion of contractual utility rather than attempting to refine the μcharts of their UIs. Interface refinement is a μCharts convention, and whilst it can be reflected in UI design in terms of change of context (which we discuss in section 7.4) it has no meaning generally within the UCD process.

The final consideration for contractual utility was that of maintaining usability. This required that we did not reduce reachability or introduce deadlock into the new UI where it was not present previously. The nature of the μcharts we build for UIs (which are essentially PIMs) means that these properties are protected by the preservation of *I_Behaviours* as it is these transitions which control movement between states of the UI such that reachability and lack of deadlock are explicitly controlled by these behaviours. We can be certain then that trace refinement will not allow such transitions to be removed as we have shown above, and as such will likewise preserve these properties.

We have shown then that the μCharts trace refinement theory can be used to formally describe our requirements for contractual utility and we can, therefore, rely on this formalism as a means of underpinning our informal refine-

ment. We have simplified our view of the UI so that is considered by way of its behaviours, which we then describe as traces within μcharts. This gives us the initially surprising results that we have shown whereby the trace refinement theory of μCharts has the same properties as our informal contractual utility. In fact, considering the UI by means of behaviour sets and describing contractual utility in terms of a subset and equality relation between those sets is what leads to this result.

6.4.2 Monotonicity and Validity

We have now developed a theory of UI refinement based on contractual utility and have subsequently shown how this can be formalised using the trace refinement theory of μCharts. We also have a way of formally describing systems and UIs together as a composition, $(UI \mid\mid Sys)$, also using μCharts. We now move on to discuss two further requirements. We want to ensure monotonicity of refinement so that we can be certain that as designers implement their UIs (relying on the process we have described) or as the system model is itself refined, we can be sure that we get a satisfactory refinement of the total $(UI \mid\mid Sys)$. We also want to ensure that the validity we have described in chapter 5 is preserved by refinement so that if we start with a valid composition, the resulting refinement is also valid.

$(UI \mid\mid Sys)$ is described using composed μcharts and Reeve has shown [80] that while the composition of μcharts is monotonic with respect to refinement, it requires three additional side-conditions to hold. The three side-conditions

are ([80], Proposition 7.1.1):

$$SC1: \quad A_\Psi \sqsupseteq_x^T C_\Psi$$

$$SC2: \quad out_A \cap \Psi_{AB} = out_C \cap \Psi_{CB}$$

$$SC3: \quad out_A \cap out_B = out_C \cap out_B$$

where $(A \parallel B)$ is the original composed chart and $(C \parallel B)$ is the new composed chart, Ψ_{AB} is the set of feedback signals for $(A \parallel B)$ and Ψ_{CB} is the set of feedback signals for $(C \parallel B)$. A has been refined to C but B remains unchanged. We will discuss the meaning of these side-conditions in the next section, but to begin with we use them as the basis for ensuring correct monotonic refinement and assuming they are satisfied consider the impact this has on validity.

In order to be sure that validity is preserved during a monotonic refinement (that is refinement which satisfies the three additional side-conditions) we consider again the validity requirements given in chapter 4 and examine possible ways in which these may be broken during refinement to show why this does not in fact occur. That is, we will show that monotonic refinement does preserve validity.

For convenience we repeat each of the validity requirements below and then explain how they are maintained either by the refinement theory (behavioural or interface) or the monotonicity side-conditions.

R1 Each *S_Behaviour/Operation* pair in the *PMR* appear as the guard/action

($S\_Behaviour/Operation$) of at least one transition in the sequential UI μchart.

In order to break this validity condition the new UI would either have to have none of the transitions which satisfy the requirement or the transition signals would have to be changed so that either the *S_Behaviour* was no longer the required guard or the *Operation* was no longer output. The first two of these also break contractual utility, and we have already shown that these do not lead to correct refinements. We therefore need consider only the removal of the *Operation* from the output of transitions. If our original UI has an *S_Behaviour* called $SBeh1$ which is related in the *PMR* to an operation called *One* then a valid μchart of the UI will have a transition with the guard $SBeh1/One$. If we remove the output from this transition so that the guard becomes $SBeh1/\{\}$ we introduce a new trace which is not present in the original (namely $(\langle\{S\_Beh1\}\rangle, \langle\{\}\rangle)$. Trace refinement therefore prevents us from making any changes which break R1 and so we can be certain that this requirement will always be preserved.

R2 Each *I_Behaviour* appears as the guard on at least one transition in the sequential UI μchart.

Breaking this requirement requires that we either remove the *I_Behaviour* from the guard of qualifying transitions, or remove the transitions themselves. We have shown already that this is not possible (as it breaks contractual utility) and so again we can be certain that this property will be preserved

during refinement.

R3 All operations related to *S_Behaviours* appear in the feedback set of the composition.

So far we have not considered feedback between composed charts as trace refinement (as related to sequential charts) does not include feedback. It may then be possible during a correct refinement to change the feedback set so that it does not contain all of the required operations. However, if we consider the second monotonicity side-condition, which is:

$$out_A \cap \Psi_{AB} = out_C \cap \Psi_{CB}$$

and also take into consideration part of refinement definition 6, which states:

$$Out_A = Out_C$$

and finally consider that the chart of the initial composition meets all of the validity requirements then we see that such a change is not possible. An example composed chart is shown in figure 6.12. Ψ_{AB} includes *only* the signals used for communication between the charts *i.e.* the *Operations* so that in fact $\Psi_{AB} = Out_A = Op1, Op2, ..., Opn$. The refinement condition that the output interfaces of A and C are the same means that we know that Out_C is also $Op1, Op2, ..., Opn$. Given side-condition two it then follows that the feedback set of the new composition $(C \parallel B)$ must contain at least all of the original

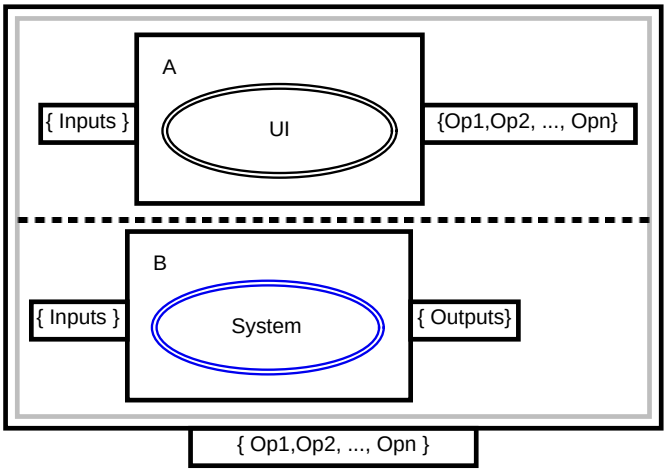


Figure 6.12: Composed Chart of Valid (UI || Sys)

signals or the side-condition will not be met (as there are no signals in this set which are not in the intersection of Ψ_{AB} and Out_A). Meeting the side-condition therefore ensures that the validity requirement is preserved.

R4 Both sequential charts in the composition must have their natural interfaces.

Behavioural refinement requires that input and output interfaces of the chart remain unchanged, therefore we need only consider the possibility that interface refinement (or a combination of interface and behavioural refinement) may break this requirement. In general μchart refinement we can increase, or decrease, interfaces of charts to change both the context that the chart is in (by controlling how it responds to the environment) as well as its external reactivity (by controlling its outputs). Given that we start with a valid chart we know that the initial interfaces are the natural interfaces of the chart (no signals are filtered), we also know that the input signals include all of the

S_Behaviours and *I_Behaviours* and that each of these are used as guards on transitions in the chart. Removing *S_Behaviours* from the input interface leads either to chaotic behaviour or to new traces of the form $S\_Beh/\{\}$ as we have shown above, and is therefore not a possible refinement. As we have already discussed, removing *I_Behaviours* from the interface has no meaning for UI refinement as the interfaces are a mechanism of μcharts rather than of the PIMs, presentation models or UIs themselves and as such we do not consider this within our refinement process (of course it is always possible for designers to take *any* actions but as we have stated at the beginning of this thesis we assume, at least, that a UCD approach is being followed rather than random behaviour). The requirement is, therefore, preserved as shown previously by the conditions on behavioural, or combined refinement.

R5 The input interface to the composition is exactly the same as the natural interface of the sequential UI μchart

As above, the inability to restrict or meaningfully expand the interface (due to refinement constraints on *S_Behaviours*, the nature of *I_Behaviours* within the μchart and the fact that the initial chart already meets this condition) ensures we can be certain this requirement will be maintained.

Having considered each of the validity conditions we can be certain that monotonic refinement will preserve validity. This is in part due to the strictness of our refinement conditions in terms of the UI behaviours as well as the validity requirements we impose on the μcharts we create for $(UI \parallel Sys)$. We

must also ensure, of course, that the addition of the side-conditions does not further restrict the possible refinement for our UIs such that UIs meeting our contractual utility requirements are not permissible refinements. We show next how the correspondence between our validity conditions and the underlying rationale of the monotonicity side-conditions prevents this from occurring.

6.4.3 Monotonic Refinement

There are two reasons that the additional side-conditions are required when we are refining composed μcharts. The first is that refinement for sequential charts allows changes to input and output signals (either through interface restriction/relaxation or behavioural changes of transitions) which may affect the communication between the composed charts. The second is that changes to the feedback set (used to determine which signals the composed charts use to communicate) are not constrained by refinement for sequential μcharts (as feedback is not used within sequential charts) and as such nothing is said about what is allowable and what is not in terms of changing signals in this set.

Side-condition one, which is:

$$A_{\Psi} \sqsupseteq_{x}^{T} C_{\Psi}$$

imposes a stricter refinement condition on charts A and C than that given by behavioural and interface refinement 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 the condition states that in this context a refinement must still hold. The superscript ^{T} relates to the correctness property of the partial relation refinement semantics given by Reeve [80] and for our purposes can be simplified to mean that all state/input pairs with defined behaviour in the original chart are retained (so the chart cannot become less reactive) and secondly that output behaviour with respect to feedback does not change (that is, output signals which are also in the feedback set must be preserved). This, of course, relates directly to one of the purposes of our validity requirements, which is to ensure correct communication between UI and system charts. This side-condition will always be met when we refine valid compositions precisely because of the strictness we impose upon the behaviours of the UI for contractual utility. We have described how this strictness limits allowable refinements and this is exactly the intention of SC1, to limit the refinements to those which preserve the reactivity of the chart as it relates to the chart it is composed with.

Side-condition two, which is:

$$out_A \cap \Psi_{AB} = out_C \cap \Psi_{CB}$$

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. We have shown above how this is useful in ensuring one

of our validity requirements is preserved. Rather than restricting possible refinements then, SC2 rather ensures correctness of refinements by guaranteeing communication is preserved.

Finally, the third side-condition, which is:

$$out_A \cap out_B = out_C \cap out_B$$

requires the intersection of output signals from each composed chart be the same. This 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. Outputs from our valid compositions are in fact used only for communication between the charts, rather than to control the external environment (we have already commented how this leads to the property where the output interface of the UI chart is the same as the feedback set). The preservation of these outputs (due to SC2 and refinement in general) means that refinement of the UI chart will automatically respect this side-condition. Of course we have no such guarantees of the system chart (which may be also communicating with other charts representing other parts of the system) so that the side-condition is necessary in ensuring that any changes made during that refinement do not adversely impact on ($UI \parallel Sys$).

The correspondence between validity, contractual utility (which is defined partly in terms of communication between UI and system) and the side-conditions ensures, therefore, that rather than creating additional restrictions

upon UI refinement, the side-conditions are either already a part of our refinement considerations (SC1 and SC3) or are necessary to guarantee communication is maintained at the feedback level (SC2) and as such do not affect individual chart refinement.

6.5 Discussion

Our reasons for wanting to find a way to describe refinement for UIs was to ensure that we have a complete design process for applications in which we can create links between the formal and informal processes from the earliest design stages through to implementation. We have shown in earlier chapters how we can create formal models of design artefacts which enable us to check for a correspondence between system and UI design once initial models have been created (which may be formal specifications, UI prototypes *etc.*) and in this chapter we have extended this to include the transformation of these models into an implementation in a structured manner (via a refinement theory).

By considering refinement generally and identifying core properties common to different kinds of refinement we were able to characterise UI refinement and develop an informal approach to refining UIs based on contractual utility. This considers both behavioural and usability properties of UIs and encompasses the interaction between both user and UI and UI and system. We have also shown how presentation models and PIMs can be used to check for contractual utility which provides a light-weight method for considering refine-

ment by designers which supports their informal process. Our intention then was to provide methods to support UI designers when they are implementing their designs which give them guarantees of correctness over and above the intuitive approach typically employed. This is not in any sense an attempt to mechanise UI development in the way of automated UI generation, but rather provide a framework for designers to work within while they exercise their design and development skills in implementing UIs.

The second part of this chapter describes how the trace refinement theory of μCharts can be used to formally support the use of contractual utility as a method of UI refinement. Using this we were able to show that such refinement is monotonic (provided the necessary side-conditions are met) and that validity is preserved during monotonic refinement. We explained the correspondence between the validity requirements and monotonicity side-conditions which leads to this property. By giving an underlying formalism for UI contractual utility we enable both UI designers and formal practitioners to work at their required level of formality whilst being assured that their different approaches have the same meaning.

One of our stated aims at the beginning of this thesis was to solve the problem of separate development of UIs and systems and in particular the problems that may occur at implementation when separately designed UIs and systems are put back together. The refinement theory given in this chapter is the final piece in our solution.

In chapter 7 we present a larger example (based on the PIMed example)

showing how our methods may be used in practice. We then present a second example showing a different refinement requirement. We then conclude this thesis with an overview of the work presented along with a discussion of potential future work.

Chapter 7

Refinement Examples

7.1 Introduction

In this chapter we return to the PIMed example and show how the models and refinement techniques we have described in the previous chapter can be used on a real-world software example. The requirements for the PIMed tool were given in section 3.3 and these, along with the additional work described in section 4.3.5 were used as the basis for the prototypes and subsequent corrections. As our first refinement example we will show how different prototypes for the PIMed system can be compared using refinement and how designers can take advantage of the light-weight contractual utility process to achieve this.

Following this example we will introduce a variant on refinement, and show how UIs for the same application, but which are designed to run on different types of hardware, may not fit into the model of refinement we have described. We explain why this is so and discuss different ways of approaching this type

of refinement.

7.2 PIMed Refinement Example

We have described in chapter 3 the PIMed prototypes which consist of a large interface (with multiple windows and dialogues). Prior to the development of these designs a different prototype was developed. This earlier prototype was consistent with the system specification, so could be considered *correct*, but was subsequently rejected as too cluttered and too confusing for users. Despite the fact that the original design consists of just three windows and the subsequent design consists of twenty seven different windows and dialogues, both designs were developed from the same requirements, and both are consistent with the underlying system. We should, therefore, be able to show, both informally and formally, that there is a refinement between the two designs. We start by showing how the informal refinement methods can be used to easily establish whether or not this is the case.

The original prototype, which we will refer to as PIMed_1 (and the subsequent prototype as PIMed_2), consists of just three windows which we show in figures 7.1, 7.2 and 7.3. The presentation models for the design are:

$$PIMed \textit{ is } PIMedMain \ : \ PMWindow \ : \ PIMWindow$$

$$PIMedMain \textit{ is } \quad (FileMenu, Container, ()),$$

$$(QuitMenuItem, ActCtrl, (QuitApp)),$$

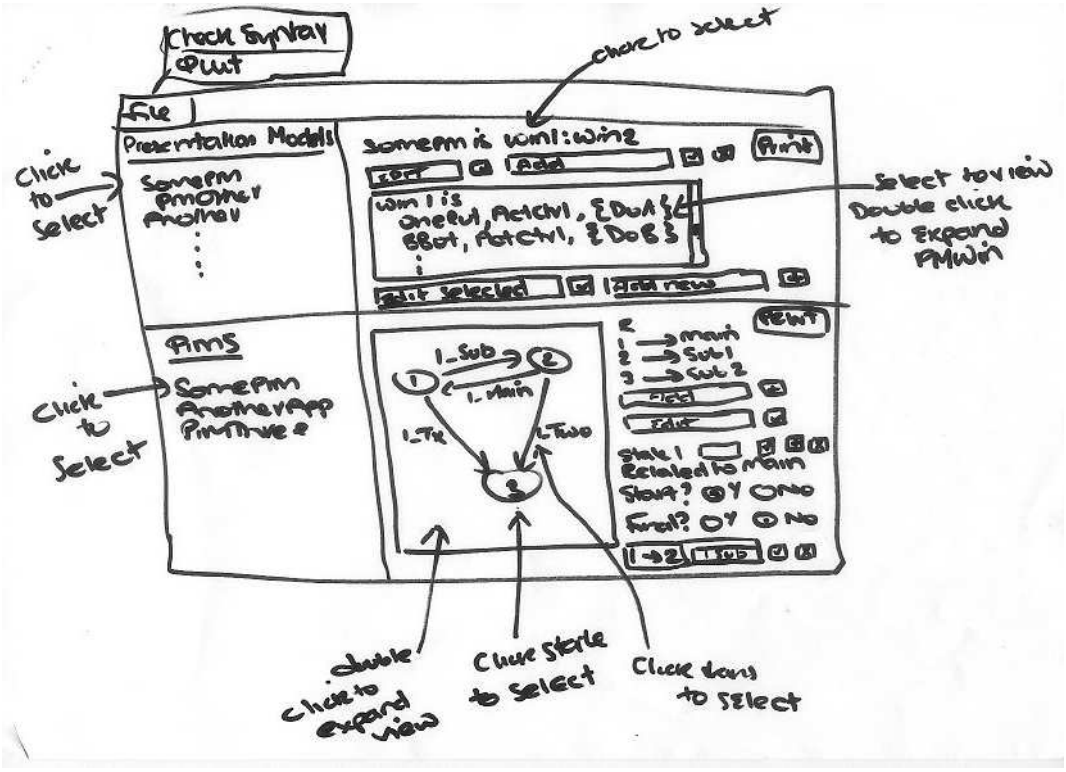


Figure 7.1: Original PIMed Design Main Screen

$(CSMenuItem, ActCtrl, (S\_CheckSyntax)),$

$(PMList, StatusDisplay, ()),$

$(PMList, SValSel, (I\_ShowPModel)),$

$(PIMList, StatusDisplay, ()),$

$(PIMList, SValSel, (I\_ShowPModel)),$

$(PIMTopDec, StatusDisplay, ()),$

$(PIMTopDec, SValSel, (I\_ShowPMLine)),$

$(PMLineEBx, StatusDisplay, ()),$

$(PMLineEBx, TxtArea, ()),$

$(PMLineTxt, TxtArea, ()),$

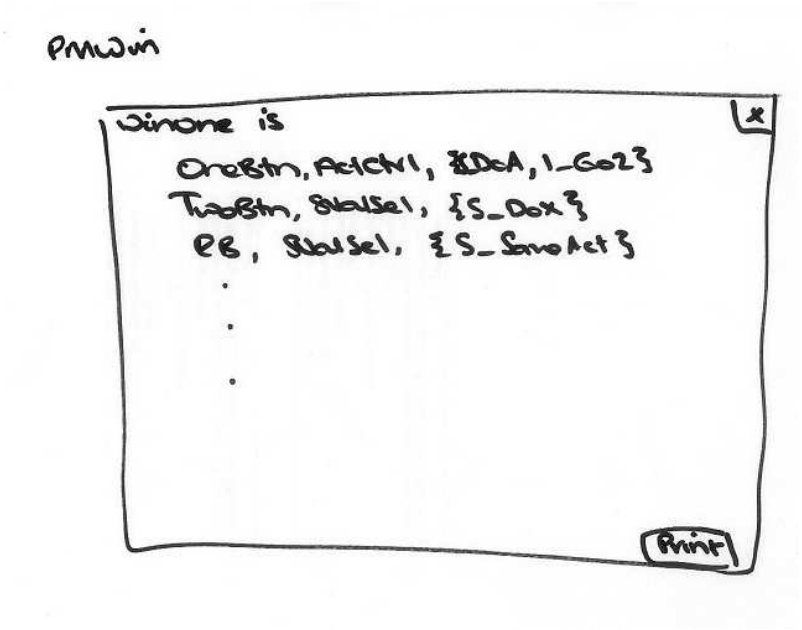


Figure 7.2: Original PIMed Design PMWindow

$(CPMEdit, ActCtrl, (S\_EditCompPMName, S\_UpdateDecs)),$

$(CPMAdd, ActCtrl, (S\_PMName, S\_UpdateDecs)),$

$(CPMDel, ActCtrl, (S\_DeleteCompPM, S\_UpdateDecs)),$

$(PrintBtn, ActCtrl, (S\_PrintPModel)),$

$(PMDetail, StatusDisplay, ()),$

$(PMDetail, ActCtrl, (I\_ShowWidget)),$

$(PMDetailDC, ActCtrl, (I\_PMWindow)),$

$(WidgetDtl, StatusDisplay, ()),$

$(WidgetDtl, TxtArea, ()),$

$(EditWidget, ActCtrl, (S\_EditWidget, S\_EditBeh,$

$S\_UpdateDecs)),$

$(NewWidgetDtl, TxtArea, ()),$

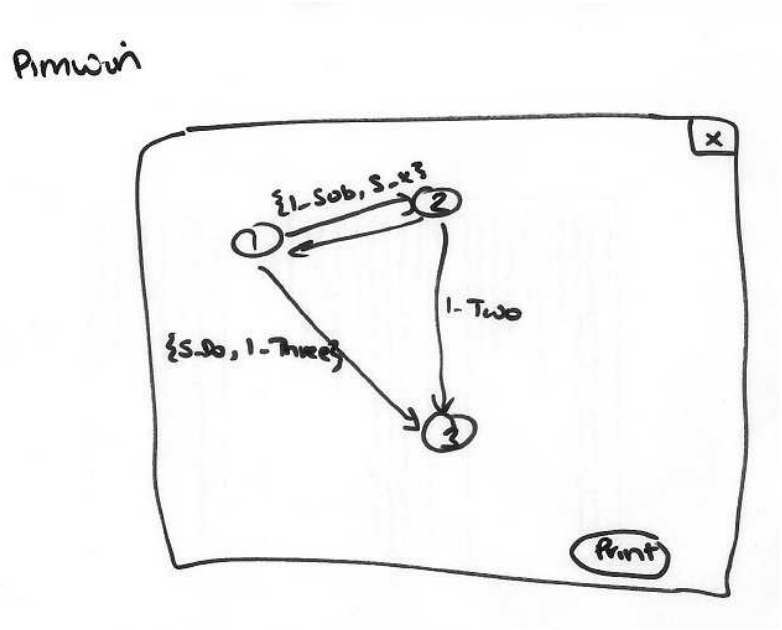


Figure 7.3: Original PIMed Design PIMWindow

$(AddWidget, ActCtrl, (S\_AddWidget, S\_AddBeh,$

$S\_UpdateDecs)),$

$(PIMDetail, StatusDisplay, ()),$

$(PIMDetailDC, ActCtrl, (I\_PIMWindow)),$

$(StateView, ActCtrl, (I\_ShowState)),$

$(PrintBtn, ActCtrl, (S\_PrintPIM)),$

$(RView, StatusDisplay, ()),$

$(RLine, ActCtrl, (I\_ShowRLine)),$

$(NewRTxt, TxtArea, ()),$

$(RTxt, TxtArea, ()),$

$(AddR, ActCtrl, (S\_AddRLine)),$

$(EditR, ActCtrl, (S\_EditRLine)),$

$(StateView, StatusDisplay, ()),$

$(StateName, TxtArea, ()),$

$(EditState, ActCtrl, (S\_EditStateName)),$

$(SStateCtrl, ActCtrl, (S\_ChangeSState)),$

$(FStateCtrl, ActCtrl, (S\_ChangeFState)),$

$(DelStateBtn, ActCtrl, (S\_DeleteState)),$

$(AddStateBtn, ActCtrl, (S\_AddState, I\_ShowState)),$

$(TransLine, ActCtrl, (I\_ShowTrans)),$

$(TransInfo, StatusDisplay, ()),$

$(TransInfo, TxtArea, ()),$

$(EditTrans, ActCtrl, (S\_EditTrans)),$

$(DelTrans, ActCtrl, (S\_DeleteTrans))$

PMWindow is

$(PrintBtn, ActCtrl, (S\_PrintPModel)),$

$(CloseBtn, ActCtrl, (I\_Main)),$

$(PMDisp, StatusDisplay, ())$

PIMWindow is

$(PrintBtn, ActCtrl, (S\_PrintPIM)),$

$(CloseBtn, ActCtrl, (I\_Main)),$

$(PIMDisp, StatusDisplay, ())$

From these we can extract the sets of *S_Behaviours* and *I_Behaviours*, which are as follows:

$$S\_Behaviours[PIMed_1] =$$

$$\{S\_CheckSyntax, S\_EditCompPMName, S\_UpdateDecs, S\_PMName,$$

$$S\_DeleteCompPM, S\_PrintPModel, S\_EditWidget, S\_AddWidget,$$

$$S\_PrintPIM, S\_AddRLine, S\_EditRLine, S\_EditStateName,$$

$$S\_ChangeSState, S\_ChangeFState, S\_DeleteState, S\_AddState,$$

$$S\_EditTrans, S\_DeleteTrans\}$$

$$I\_Behaviours[PIMed_1] =$$

$$\{I\_ShowPModel, I\_ShowPMLine, I\_ShowWidget, I\_PMWindow,$$

$$I\_PIMWindow, I\_ShowState, I\_ShowRLine, I\_ShowState,$$

$$I\_ShowTrans, I\_Main\}$$

In this example we are not looking at transforming a design into an implementation, but rather looking at the correspondence between two different designs for the same UI which were developed at different times. The designs of PIMed_2 are a revision of PIMed_1 and are close to our final implementation and therefore constitute a refinement step. Before we can compare the sets of behaviours of the two different designs we need to consider that the names chosen for *S_Behaviours* may not be the same in the presentation models of the two different designs. This is also likely to be the case where different

designers are working on the same system as there is no guarantee that they will choose the same names. However, we can identify which *S_Behaviours* are the same based on the *PMR*. So, for example, in the *PMR* of the original design there is a relation:

$$S\_ChangeFState \mapsto UpdateFinalStates$$

In the *PMR* of PIMed_2 is the relation:

$$S\_EditFinal \mapsto UpdateFinalStates$$

From this we determine that *S_ChangeFState* and *S_EditFinal* are the same behaviour (as they are related to the same underlying system operation). We then use this information to rename behaviours in one of the presentation models (in this example we will rename the original design's behaviours) before we compare the two sets of *S_Behaviours*. After renaming the set of *S_Behaviours* of the original design is:

$$\{S\_DoSyntaxCheck, S\_UpdatePMName, S\_UpdateDecs, S\_AddPMName,$$
$$S\_RemoveCompModel, S\_PrintPModel, S\_UpdateWidget,$$
$$S\_UpdateBehName, S\_AddWidget, S\_AddBehName, S\_PrintPIM,$$
$$S\_AddRPair, S\_EditR, S\_EditState, S\_EditStart, S\_EditFinal,$$
$$S\_DeleteState, S\_AddNewState, S\_EditTrans, S\_DeleteTrans\}$$

Now we can compare this with the set of *S_Behaviours* from the PIMed_2 design. Recall that we obtain the behaviours for the total UI by taking the union of the sets of behaviours from each of the component models (where each window and dialogue has its own component model), which gives:

$$
\begin{aligned}
&S\_Behaviours[PIMed_2] = \\
&\{S\_PrintPModel, S\_EditState, S\_EditTrans, S\_EditStart, \\
&S\_EditFinal, S\_EditR, S\_PrintPIM, S\_AddNewState, \\
&S\_DeleteState, S\_AddRPair, S\_DeleteTrans, S\_AddWidget, \\
&S\_RemoveCompModel, S\_UpdateDecs, S\_UpdateWidget, \\
&S\_UpdatePMName, S\_UpdateBehName, S\_AddPMName, \\
&S\_AddBehName, S\_DoSyntaxCheck\}
\end{aligned}
$$

The sets of *S_Behaviours* for the two designs are the same, so our first requirement for contractual utility, that $PIMed_1 \equiv_{SBeh} PIMed_2$, is met. We now move on to consider the *I_Behaviours*. Again we might expect that *I_Behaviours* which are the same may have different names in the two designs. In this case we cannot rely on the *PMR* to inform us which are the same, but rather we must use manual inspection to determine this. Again we will rename the *I_Behaviours* of the original design to match those of the second design which

gives us the set:

$\{I\_OpenEditCompPModel, I\_OpenEditPModel, I\_OpenEditWidget,$
$I\_OpenViewPM, I\_OpenViewPIM, I\_OpenEdStateWin, I\_OpenEdRWin$
$I\_OpenEdTransWin, I\_Main\}$

As we have done with the $S\_Behaviours$ we now compare this set of $I\_Behaviours$ with those of the second design, which are:

$I\_Behaviours[PIMed_2] =$
$\{I\_OpenAddPMDecs, I\_OpenAddPIM, I\_MaxWindow, I\_MinWindow,$
$I\_OpenViewPM, I\_OpenViewPIM, I\_OpenEditWidget, I\_OpenEditBeh,$
$I\_OpenAddPMNExst, I\_OpenAddWNExst, I\_OpenAddBExst,$
$I\_CloseViewPModel, I\_OpenEditPModel, I\_OpenEditCompPModel,$
$I\_CloseViewPIM, I\_OpenEdStateWin, I\_OpenEdRWin, I\_OpenEdSSWin,$
$I\_OpenEdFSWin, I\_OpenEdTransWin, AddNewState, I\_OpenDelSCheck,$
$I\_CloseDelSCheckWin, I\_CloseEdSSWin, I\_CloseEdFSWin,$
$I\_CloseEdRWin, I\_OpenCheckTransDel, I\_CloseEdTransWin,$
$I\_CloseCheckTDel, I\_CloseEdSWin, I\_OpenRemWidget,$
$I\_OpenAddWidget, I\_CloseEdPModelWin, I\_CloseAddW,$
$I\_OpenAddCompModel, I\_OpenRemCompModel, I\_OpenDecWarning,$

$I\_CloseRemComp, I\_CloseAddComp, I\_CloseDecWarning,$

$I\_CloseAddPMDecs, I\_CloseEditWName, I\_CloseEditPMName,$

$I\_CloseEditBName, I\_CloseAddPMNExst, I\_CloseAddWNExst,$

$I\_CloseAddBExst, I\_OpenSyntaxMsgWin, I\_CloseAddPIM$

$I\_CloseSyntaxWin, I\_Main\}$

Again we see that our contractual utility requirement, which is $I\_Beh[PIMed_1] \subseteq I\_Beh[PIMed_2]$ is met.

Based on these comparisons of behaviour sets of the two designs we are then satisfied that the new version of the design is a correct refinement of the first. Performing this sort of check using such a light-weight method (even if the sets are very large it is not a difficult or arduous task) gives UI designers the ability to adopt the formal process of refinement with little overhead. This is possible because we can, as we have shown in the previous chapter, rely on the underlying formal theory to support this light-weight method.

From the presentation models of each of the designs we can develop *PMRs* which show that they are both consistent with the underlying system specification. We can then develop the PIMs for each design which can then be used as the basis for constructing valid (UI || Sys) μcharts for each design. Figure 7.4 shows the (PIMedUI_1 || PIMedSystem) μchart. The chart for PIMed_2 is necessarily more complex due to the number of windows and dialogues involved. Recall that with μcharts we can use a decomposition syntax where we embed charts within states of other charts, which gives a modular view. The top level

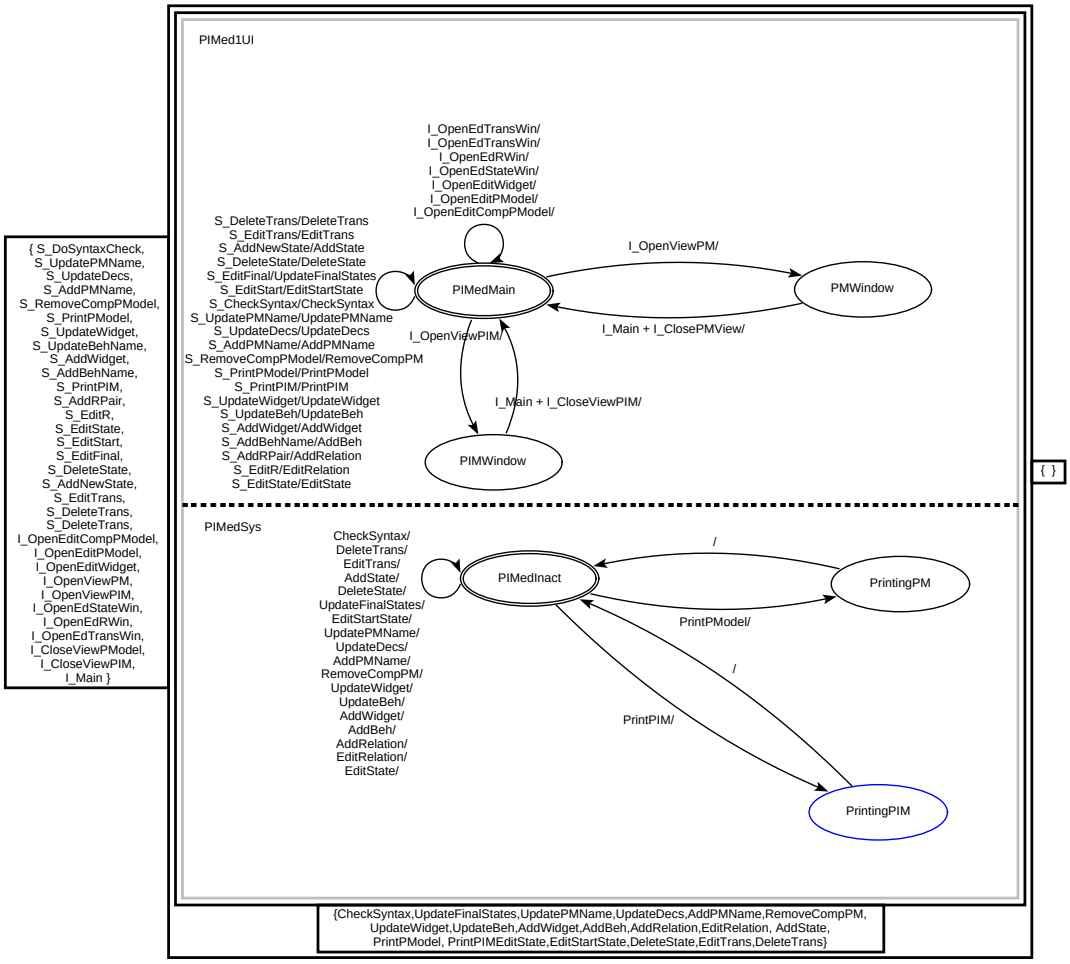


Figure 7.4: PIMed$_1$UI and PIMedSystem

μchart for (PIMedUI$_2$ || PIMedSystem) is given in figure 7.5. The states designated by rectangular boxes are themselves μcharts such that when PIMedUI$_2$ is in a decomposed state the embedded chart is active. This allows us to use trace refinement in exactly the same way as we have for the simpler charts we have presented in chapter 6 relying on μChart's semantics to unwind the meaning of each decomposed chart. All of the required signals exists within

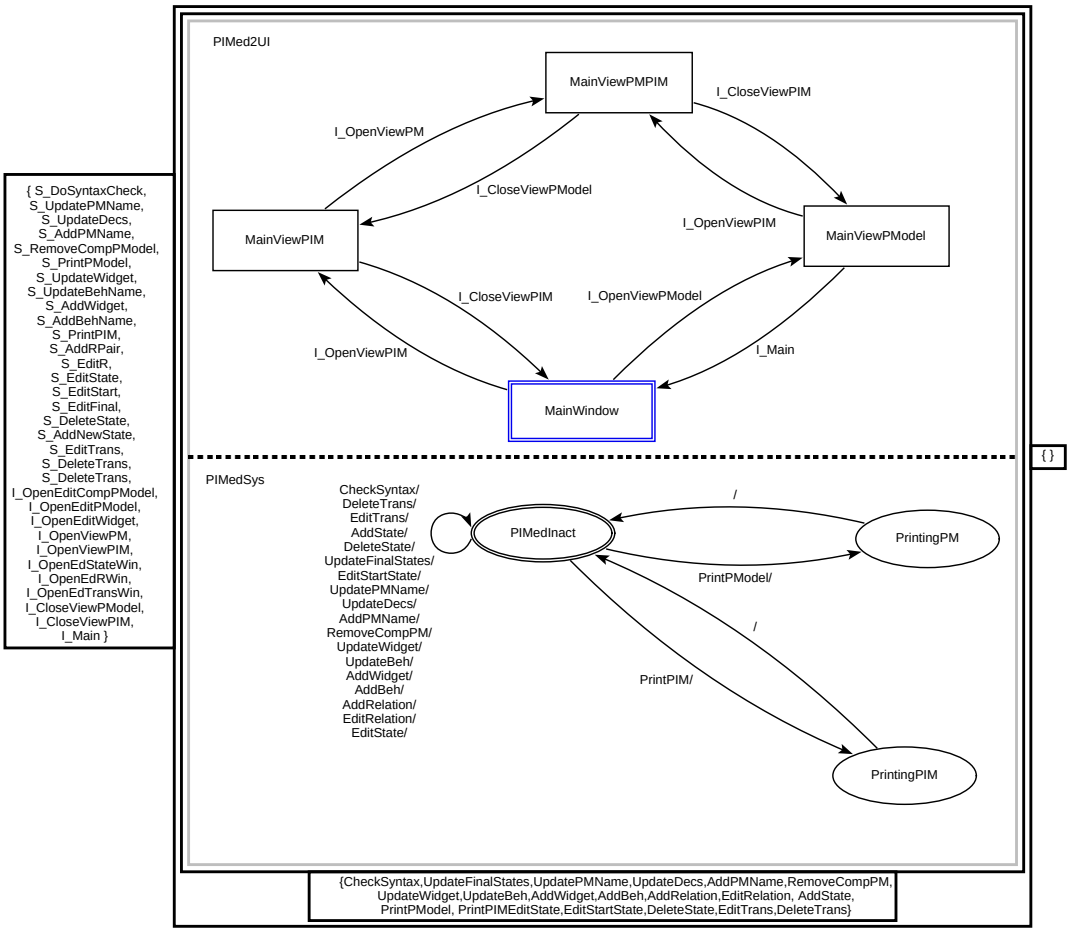


Figure 7.5: PIMed$_2$UI and PIMedSystem

the decompositions (and in some cases decompositions within those decompositions) such that the interfaces and feedback sets for both of the compositions are the same and the necessary conditions on traces for refinement are met.

7.3 PIMed Implementation

In a similar manner we can consider refinement of the PIMed_2 design to an implementation and be certain that it is correct as long as the S and I behaviour properties are respected. If we look at prototypes for two of the windows of the final design, given in figures 7.6 and 7.7, along with their implementations as horizontal prototypes (a horizontal prototype is a computer-based prototype with partial functionality) given in figures 7.8 and 7.9 we can see that the similarities between them are such that intuitive refinement can provide much of the reassurance we need. As ever we use the formal theory (or rather contractual utility supported by the formal theory) to ensure that such intuition is correct.

$S\_Behaviour$ and $I\_Behaviour$ sets for each of these are:

$$S\_Behaviours[Main] = \{\}$$

$$S\_Behaviours[MainHP] = \{\}$$

$$S\_Behaviours[AddDecWin] = \{S\_UpdateDecs\}$$

$$S\_Behaviours[AddDecWinHP] = \{S\_UpdateDecs\}$$

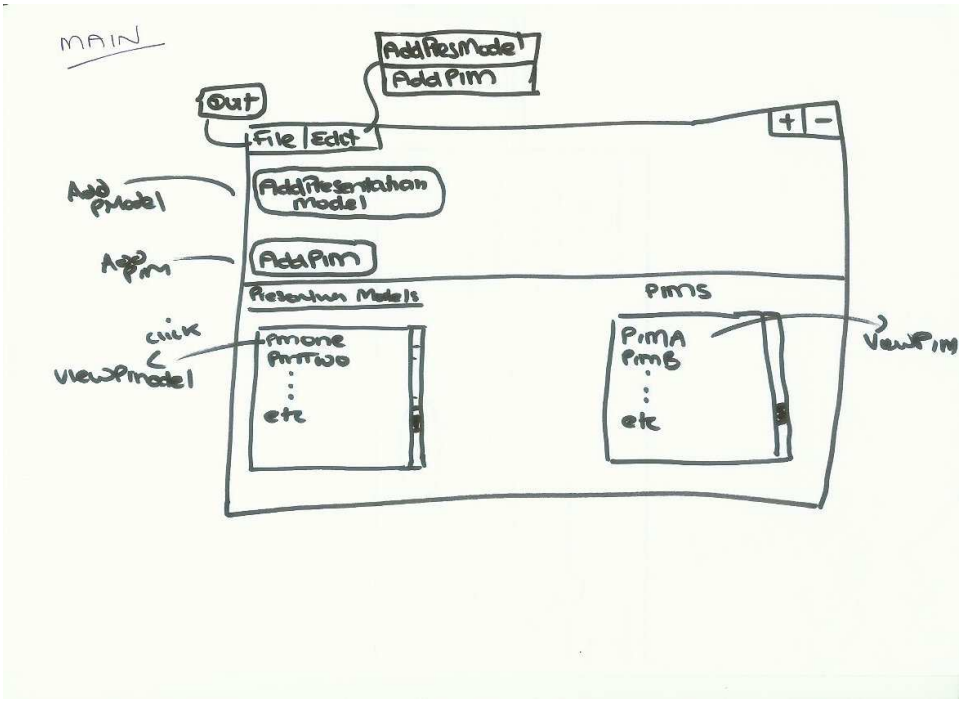


Figure 7.6: Main UI for PIMed

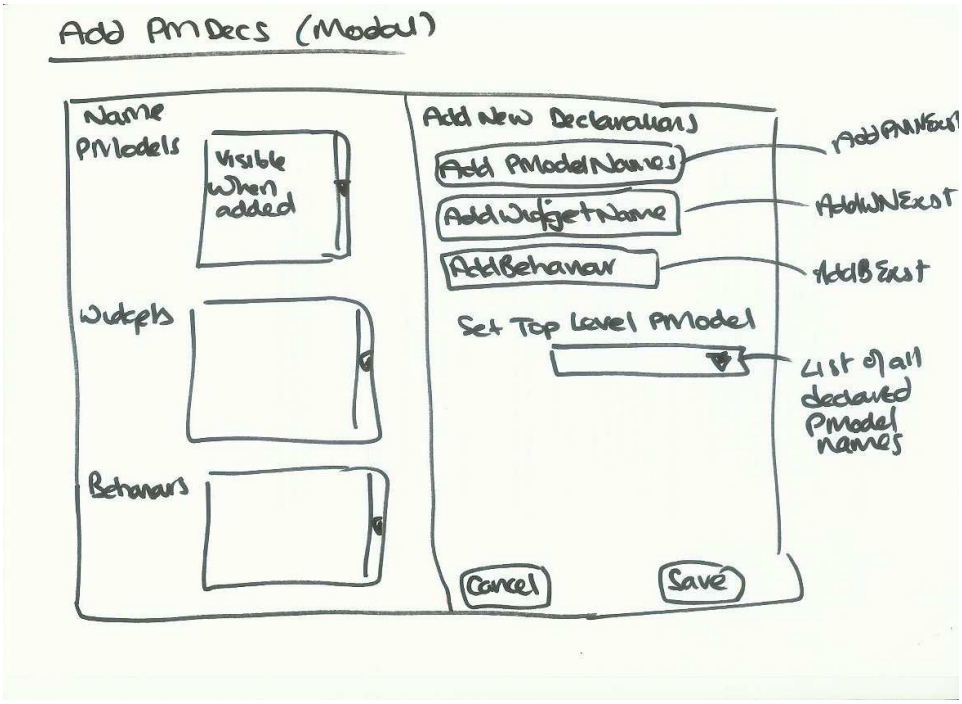


Figure 7.7: Add Declarations Window

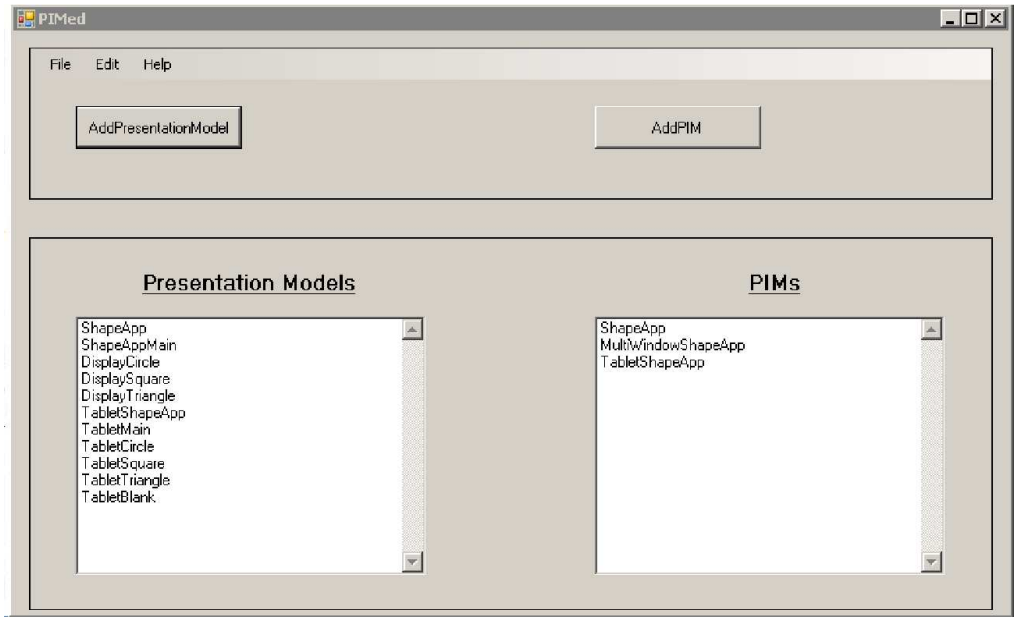


Figure 7.8: Main UI for PIMed

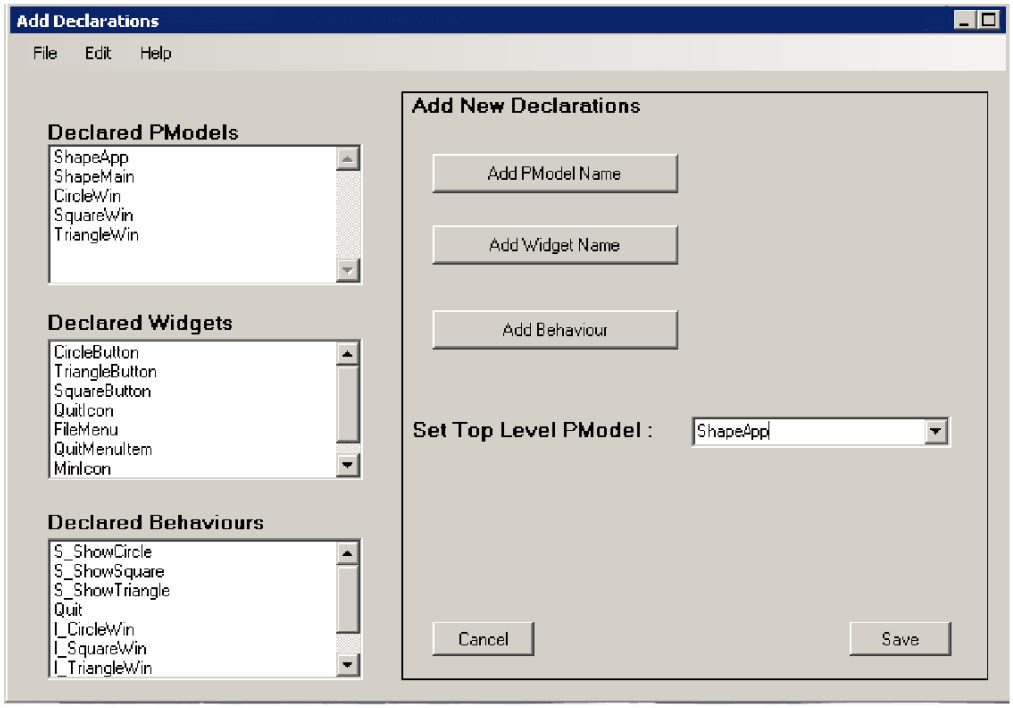


Figure 7.9: Add Declarations Window

$$I\_Behaviours[Main] = \{I\_OpenAddPMDecs, I\_OpenAddPIM,$$
$$I\_MaxWindow, I\_MinWindow, I\_OpenViewPM, I\_OpenViewPIM\}$$
$$I\_Behaviours[MainHP] = \{I\_OpenAddPMDecs, I\_OpenAddPIM,$$
$$I\_MaxWindow, I\_MinWindow, I\_OpenViewPM, I\_OpenViewPIM\}$$
$$I\_Behaviours[AddDecWin] = \{I\_OpenAddPMNExst,$$
$$I\_OpenAddWNExst, I\_OpenAddBExst, I\_CloseAddPMDecs\}$$
$$I\_Behaviours[AddDecWinHP] = \{I\_OpenAddPMNExst,$$
$$I\_OpenAddWNExst, I\_OpenAddBExst, I\_CloseAddPMDecs\}$$

From these we see that the contractual utility requirements are met for these particular windows, that is the sets of both *S_Behaviours* and *I_Behaviours* of the corresponding windows are the same. We can repeat this process for each of the window/dialogue prototypes and their respective horizontal prototypes and then finally repeat with the total behaviour of the UI (the union of all sets of component models). This shows that contractual utility is met in each case and our theory, therefore, supports the intuitive refinement.

7.4 Vertical Refinement

Now that we have an understanding of what refinement for UIs is (based on contractual utility) and its underlying formal refinement theory which is monotonic with respect to ($UI \parallel Sys$), we conclude this chapter by describing another type of refinement and explaining the differences between this and the

refinement we have described up to this point.

Our refinement considerations so far have been based on the premise that we are transforming designs into implementations, that is we are taking some abstract description of the UI (and system) and refining this into something more concrete until eventually we have an implementation. 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 the requirements (and user requirements) for both of the systems are the same, the nature of the different platforms, or the extensions required, mean that it is not immediately obvious that we can rely on the relationship between respective I and S behaviours from the presentation models in the same way that we have for horizontal refinement, as we will show shortly. In our next example we show that when we examine the models of UIs in cases where underlying hardware is different we may find ourselves in a situation where contractual utility does not appear to hold. The design of the UIs

may be very different (due to hardware constraints and differing interaction techniques *etc.*) meaning there is no direct correspondence between the presentation models (or more importantly the behaviours within the presentation models) of the different designs.

As an example we return again to the shape application. Our original version had a UI design prototype which we now repeat in figure 7.10. An

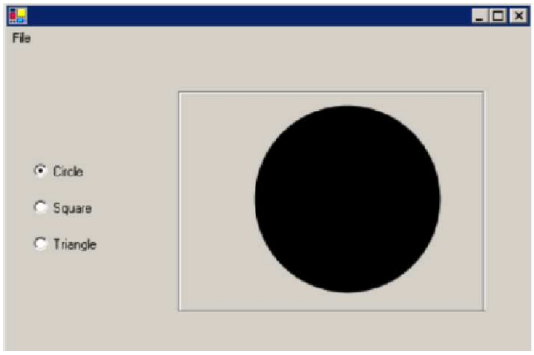


Figure 7.10: Shape Application

alternative design for this application is given in figure 7.11. Although this is a UI for the same application this time it is 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 the UIs for the two versions.

The presentation models for the UI design of figure 7.10 is:

ShapeUI is MaxUI : *MinUI*

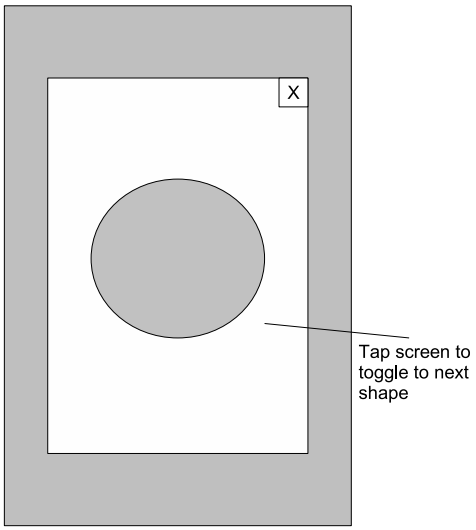


Figure 7.11: Shape App. for PDA

MaxUI is

$(CircleCtrl, ActCtrl, (S\_ShowCircle)),$

$(SquareCtrl, ActCtrl, (S\_ShowSquare)),$

$(TriangleCtrl, ActCtrl, (S\_ShowTriangle)),$

$(ShapeFrame, SValResponder, (S\_ShowCircle, S\_ShowSquare,$

$S\_ShowTriangle)),$

$(FileMenu, Container, ()),$

$(QuitMenuItem, ActCtrl, (Quit)),$

$(QuitIcon, ActCtrl, (Quit)),$

$(MinWin, ActCtrl, (I\_MinWin)),$

$(MaxWin, ActCtrl, (I\_MaxWin))$

MinUI is

$(CircleCtrl, ActCtrl, (S\_ShowCircle)),$

$(SquareCtrl, ActCtrl, (S\_ShowSquare)),$

$(TriangleCtrl, ActCtrl, (S\_ShowTriangle)),$

$(ShapeFrame, SValResponder, (S\_ShowCircle, S\_ShowSquare,$

$S\_ShowTriangle)),$

$(FileMenu, Container, ()),$

$(QuitMenuItem, ActCtrl, (Quit)),$

$(QuitIcon, ActCtrl, (Quit)),$

$(MinWin, ActCtrl, (I\_MinWin)),$

$(MaxWin, ActCtrl, (I\_MaxWin))$

and for figure 7.11 is:

PDAShapeUI is

$(QuitIcon, ActCtrl, (Quit)),$

$(ActiveArea, ActCtrl, (S\_ToggleShape)),$

$(ActiveArea, SValResponder, (S\_ToggleShape))$

In order to consider whether or not a refinement holds between the two designs we examine the sets of behaviours from each of these models to determine

whether contractual utility holds.

$$S\_Behaviours[ShapeUI] = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$$

$$I\_Behaviours[ShapeUI] = \{I\_MinWin, I\_MaxWin\}$$

$$S\_Behaviours[PDAShapeUI] = \{S\_ToggleShape\}$$

$$I\_Behaviours[PDAShapeUI] = \{\}$$

The sets of *S_Behaviours* of the two presentation models are not equivalent, and as such we would state that contractual utility, and therefore refinement, does not hold. The problem appears to be that *ShapeUI* has more behaviours than *PDAShapeUI*. However when we consider the requirements for the two applications then we know that this is not the case for the required functionality (both systems being based on the same set of requirements). At this level of abstraction (where we model the intended requirements as behaviours) we therefore expect that both systems should have the same behaviours. The nature of interaction with the PDA device, however, means that rather than there being a discrete behaviour to display each different shape in the PDA application there is a single behaviour which is repeated a number of times depending on which shape is to be displayed. In this example rather than there being a one-to-one correspondence between the *S_Behaviours* of the two UI presentation models there is instead a relationship between a single behaviour of *ShapeUI* and several (repeated) behaviours of *PDAShapeUI*.

This situation is not unusual in general refinement theory based on sim-

ulations, where two systems have a relation described between them (often called a *retrieve* relation) which is then shown to hold when corresponding operations occur. In [30] Derrick and Boiten describe how cases such as our example, where an abstract system has a single operation and the concrete system refines this to several operations (which they refer to as non-atomic refinement) can be dealt with. One approach they present is via the use of stuttering steps where a single operation of an abstract system model is refined by several operations in a concrete model by introducing a skip operation (an operation where the state does not change) in the abstract model.

Figure 7.12 shows how displaying a triangle as the first action of the *ShapeUI* can be described as *S_ShowTriangle*, *Skip*, *Skip* in the *ShapeUI* model and as *S_ToggleShape*, *S_ToggleShape*, *S_ToggleShape* in the *PDAShape* model. There is, however, a problem with taking this approach. At each step (*i.e.* af-

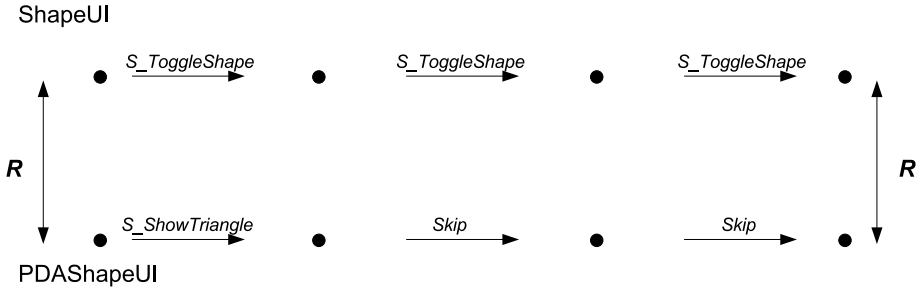


Figure 7.12: Stuttering Steps

ter each atomic operation) the retrieve relation between the two systems must still hold. In our example where we would consider the retrieve relation to be between the currently displayed shape of each UI (so that if *ShapeUI* shows a

triangle then *PDAShapeUI* also shows a triangle) it is not the case that the relation is preserved at each step, figure 7.13 shows why this is so.

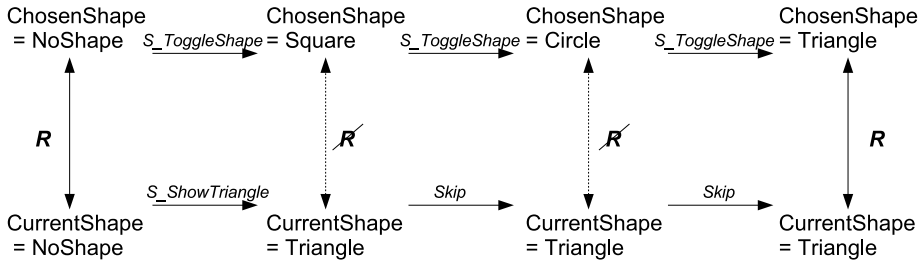


Figure 7.13: Stuttering Steps, Skip and R

There is an additional problem in that we cannot state in general how many *skip* operations are required to relate each atomic operation of *ShapeUI* with the operation of *PDAShape*, it depends on the start state of *PDAShape*. For example, when the systems are initialised both show no shape. A single *S_ToggleShape* behaviour leads to a square being displayed, so in this case there is a one-to-one correspondence with the *S_ShowSquare* behaviour. However when the systems are in a state where a circle is being displayed, then *PDAShape* requires two *S_ToggleShape* actions to display a square again, whereas *Shape* still only requires the single *S_ShowSquare* behaviour. Derrick and Boiten [30] describe non-atomic refinement only where an operation gets decomposed into a sequence of operations of fixed length rather than when the length is variable as is the case with our example. Their reasoning is that allowing their non-atomic refinement method to deal with sequences of operations whose length is state-dependent would require their refinement theory to become much more complicated in order to deal with the possible com-

plications due to partiality and unbounded nondeterminism[1], and so they do not attempt to describe refinement for such cases. While our example does not lead to unbounded nondeterminism (we have a fixed number of operations related to each state), because the length of the sequence of operations is not fixed we cannot follow Derrick and Boiten's refinement approach.

While this is not then a suitable approach to take for such refinement cases it does give us some idea of how to proceed, in that it highlights the need to consider not only the operations of the UI but also the state. If we now look at the system specification and consider the relationship between the two different UIs to the system this becomes clearer, especially when we try to create the *PMR* between each of the presentation models and the system specification.

The system specification for shape application can be described, using Z, as follows:

$$SHAPE ::= Circle \mid Square \mid Triangle \mid NoShape$$

ShapeSystem

$$currentShape : SHAPE$$

[1]unbounded because it is impossible to determine how many such operations may be required

InitShape

$ShapeSystem'$

$currentShape' = NoShape$

DrawCircleOperation

$\Delta ShapeSystem$

$currentShape' = Circle$

DrawSquareOperation

$\Delta ShapeSystem$

$currentShape' = Square$

DrawTriangleOperation

$\Delta ShapeSystem$

$currentShape' = Triangle$

Now, if we describe the relations between behaviours of presentation models and specification we get the *PMR* shown in figure 7.14.

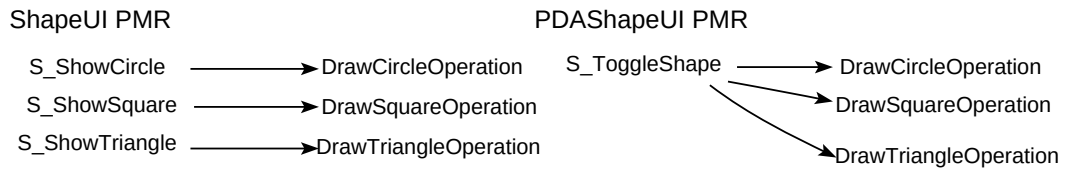


Figure 7.14: Shape UI PMRs

The *PMR* for *PDAShapeUI* does not meet the requirement of a total many-to-one relation (it is in fact one-to-many) and suggests nondeterminism as we cannot say for certain which system operation *S_ToggleShape* will invoke. There appears to be an inconsistency between the UI design and the underlying system. However, the design itself is not nondeterministic (shapes do not appear randomly) but rather the behaviour associated with the user interaction of tapping the screen is determined by the current state of the system. The presentation model is in this case too abstract as it only describes part of the the intended meaning of the design. In order to correctly describe the UI behaviour we must somehow include the current state of the system.

We have already seen a way of describing state within presentation models when we have considered UIs with multiple windows (which are themselves different states of the UI). *I_Behaviours* are used to move between different UI states and we can use this approach to expand our description of *PDAShapeUI* by treating each visible state of the UI as a different window and describing them independently. We can then create an expanded presentation model to describe this as follows:

$PDAShapeUI\ is\ PDABlank\ :\ PDASquare\ :\ PDACircle\ :\ PDATriangle$

$PDABlank\ is$

$(QuitIcon, ActCtrl, (Quit)),$

$(ActiveArea, SValResponder, (S\_ShowSquare, I\_PDASquare))$

$PDASquare\ is$

$(QuitIcon, ActCtrl, (Quit)),$

$(ActiveArea, SValResponder, (S\_ShowCircle, I\_PDACircle))$

$PDACircle\ is$

$(QuitIcon, ActCtrl, (Quit)),$

$(ActiveArea, SValResponder, (S\_ShowTriangle, I\_PDATriangle))$

$PDATriangle\ is$

$(QuitIcon, ActCtrl, (Quit)),$

$(ActiveArea, SValResponder, (S\_ShowSquare, I\_PDASquare))$

Now when we create the PMR between this presentation model and the system we get:

$S\_ShowSquare \mapsto ShowSquareOperation$

$S\_ShowCircle \mapsto ShowCircleOperation$

$S\_ShowTriangle \mapsto ShowTriangleOperation$

This shows how we can ensure that the presentation model correctly describes

the design by including considerations of state by way of *I_Behaviours*. However, if we now return to the comparison of behaviours of the two UI designs, which are now:

$$S\_Behaviours[ShapeUI] = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$$

$$I\_Behaviours[ShapeUI] = \{I\_MinWin, I\_MaxWin\}$$

$$S\_Behaviours[PDAShapeUI] = \{S\_ShowCircle, S\_ShowSquare, S\_ShowTriangle\}$$

$$I\_Behaviours[PDAShapeUI] = \{I\_PDASquare, I\_PDACircle, I\_PDATriangle\}$$

The two designs have equivalent sets of *S_Behaviours*, but including the state of the PDA UI by way of *I_Behaviours* means that

$$I\_Behaviours[ShapeUI] \not\subseteq I\_Behaviours[PDAShapeUI]$$

$$I\_Behaviours[PDAShapeUI] \not\subseteq I\_Behaviours[ShapeUI]$$

and so based on our earlier definition contractual utility still does not hold.

Given the different types of interaction and underlying hardware for the application it is not surprising that the *I_Behaviours* (which are directly related to interaction and control of the UI) do not correspond between the two designs. It appears then that our contractual utility requirement for

I_Behaviours is too strict for vertical refinement. Of course not all examples will exhibit this problem (which in this example is an artefact of the *type* of underlying hardware and interaction method), but in general we can expect similar problems to occur for the reasons outlined above.

If we choose to remove the requirement on *I_Behaviours* for vertical refinement then we can no longer rely on μChart's trace refinement as the underlying formal theory. Figure 7.15 shows the μchart for *ShapeUI* and figure 7.16 the μchart for *PDAShapeUI*. If we try to show that the *PDAShapeUI* chart is

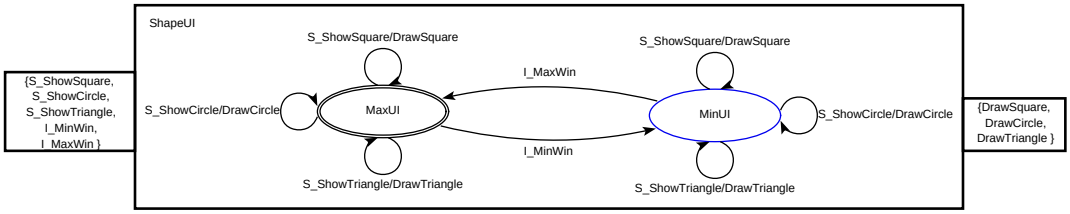


Figure 7.15: Shape UI μchart

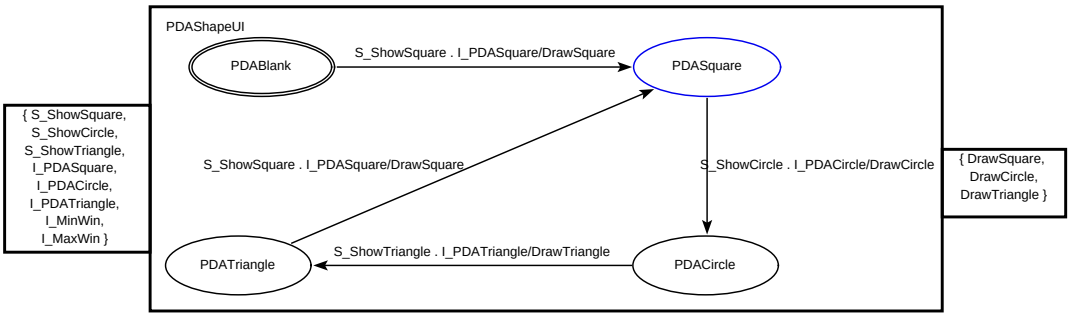


Figure 7.16: PDAShape UI μchart

a refinement of *ShapeUI* chart we cannot, as there are traces of *PDAShapeUI* which are not traces of *ShapeUI* due to inputs of *I_MinWin* and *I_MaxWin* leading to chaotic behaviour in *PDAShapeUI*. As we have explained in sec-

tion 6.4.2 we cannot remove these signals from the input interface to prevent this chaos from occurring. Conversely, if we try instead to prove that *ShapeUI* is a refinement of *PDAShape* we have a similar problem with the signals *I_PDASquare*, *I_PDACircle* and *I_PDATriangle*.

Whilst it appears that trace refinement, in this instance, is not a helpful formalism, if we examine the two charts again we see that it does in fact suggest a solution. Refining *ShapeUI* to *PDAShapeUI* is problematic because the chart has no defined behaviour for signals *I_MinWin* and *I_MaxWin*, however if we introduce these signals to the *PDAShapeUI* chart we can solve this problem. What we would like is for the signals to be ignored, but rather than refining the interface to achieve this (which we have already discussed and shown will not succeed) we can instead create silent transitions (loop transitions with no output) on each of the states of *PDAShapeUI* which have *I_MinWin* or *I_MaxWin* as their guard. This is similar to the notion of *skip* we described earlier, but in this case we introduce transitions which behave in the same way as *skip* into μcharts, *i.e.* transitions which do nothing, and which (more importantly) preserve the relation on states between the two charts. The result of this is the μchart given in figure 7.17. We can then show that this new version of *PDAShapeUI* is a refinement of *ShapeUI* but we must rely on the formal theory to do this rather than contractual utility.

In order to consider vertical refinement by way of contractual utility we must then exclude the requirement on *I_Behaviours*. As we have shown it is not sensible to expect that the interaction possibilities relating to the interface

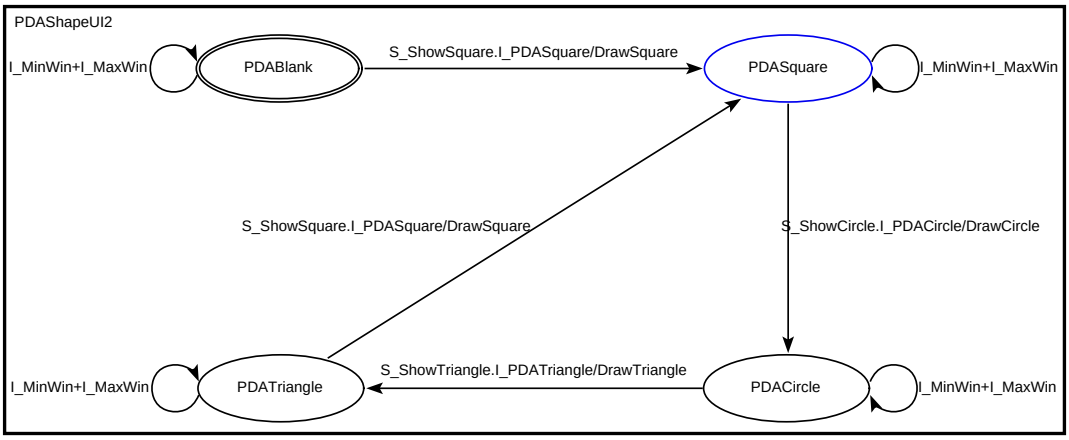


Figure 7.17: PDAShape UI with Silent Transitions μchart

navigation be the same on systems whose interfaces may have no similarity in terms of their interaction methods. For vertical refinement we then have a weaker version of contractual utility which relies solely on $S\_Behaviours$. It is easy to remove the strictness of our initial conditions for contractual utility from the informal process because there are no additional constraints to consider; this is not the case for the formal theory however. In order to support this weaker version formally, we must however identify the $I\_Behaviours$ which are no longer relevant to the interaction and add them as *skip* transitions to the μchart in the manner shown above.

This weaker version of contractual utility gives us a general method which we can use for vertical refinement cases. However, while it is useful and important for ensuring behavioural properties are preserved, by disregarding the conditions on $I\_Behaviours$ we lose the ability to consider interactivity and preservation of usability. For example, we have shown in chapter 6 how main-

taining the set of *I_Behaviours* ensures any refinement will provide a least the same UI navigation methods to users as the original. We cannot make any such claims for this weaker version of refinement. While it is useful, therefore, to have a way of considering vertical refinement in this manner, we depend on our original, stronger, requirements for refinement in general.

7.5 Discussion

In this chapter we have shown, by example, how contractual utility can be used to easily consider refinement between UI designs, different types of prototypes, and implementations, for non-trivial systems. Whilst the underlying μchart representations become increasingly complex as UI size and complexity increases this is not the case for contractual utility. For UI designers then the refinement process is straightforward and not limited to simple or small UIs.

We have also introduced a second type of refinement, vertical refinement and shown (again by example) how this may lead to problems with contractual utility due to different hardware or interaction requirements. We have presented a solution to this which requires a weakening of our definition of contractual utility and the addition of silent transitions to the μcharts to ensure that the formal theory still supports the informal methods.

Chapter 8

Conclusions

8.1 Overview

In the introduction to this work we stated our hypothesis that formal methods and informal user-centred design methods could be used together in an integrated manner. This thesis has presented an approach which provides such an integration and has further shown the benefits of such an approach. Figure 8.1 presents an overview of how the new methods we have proposed fit into existing UCD and formal methods devlopment processes.

The contributions of this thesis are as follows. We have given the derivation and description of two new models, the presentation model and the PIM, which can be used to formally describe informal UI design artefacts, and shown by example how these can be used to create a link between UI design and formal system specification, as well as provide benefits to the UI design process itself.

We have subsequently discussed how we can develop the link further by us-

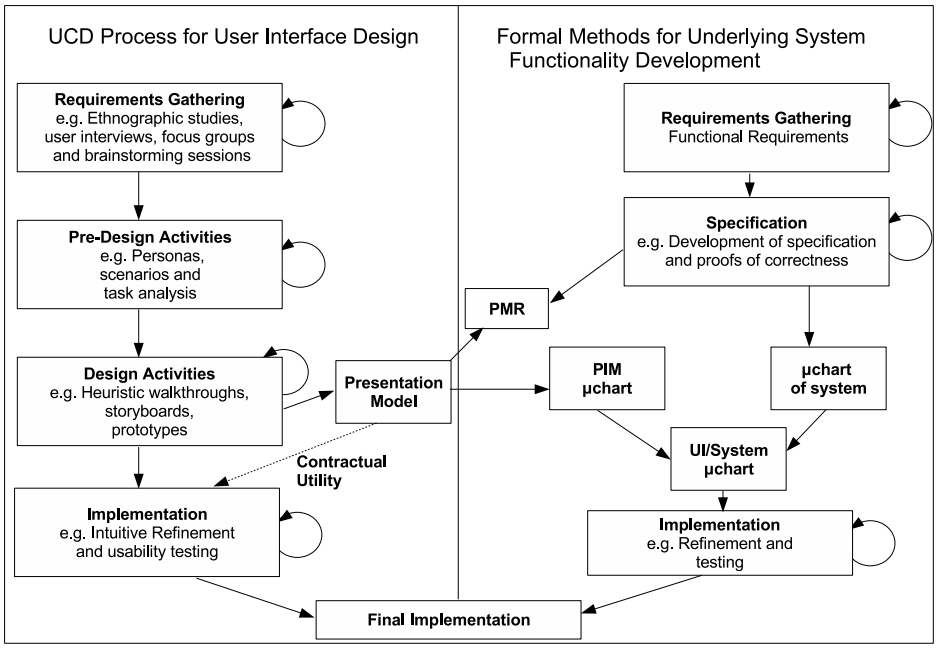


Figure 8.1: Structure and Responsibilities for New Process

ing the μCharts language to describe the composition of the UI and system and have given a definition of valid applications (consisting of such a composition) as well as the requirements on μcharts for describing such valid applications. Finally we have discussed the importance of refinement for UIs in the context of our work, and from a general understanding of refinement principles derived a refinement theory for UIs which has both a light-weight description by way of contractual utility and a formal underlying theory given by the trace refinement theory of μCharts. The formal theory allows us to show that refinement is monotonic with respect to the composition of UI and system and that monotonic refinement preserves validity. We have also discussed additional uses for refinement by way of vertical refinement and described how a weaker version of contractual utility can still be supported formally to support this.

We have, therefore, developed a new process for UI designers which can be integrated into their UCD process with very little overhead (in terms of both learning and time). The design process begins in the usual manner by developing designs and prototypes based on user requirements, task analysis, perhaps ethnographic studies (or whatever their initial methods are). We now provide links between these early designs and the formal system specifications which enable designers (or formal specialists, or both) to ensure consistency between the designs and the system. The designers have the additional work of developing the presentation models and PIMs but this is not time-consuming and does not require that they learn a complex formalism, and as we have shown in chapter 4, the act of building such models is beneficial in itself. The models do not require additional knowledge (of the underlying system for example) but make explicit the design decisions already made and describe the understanding that the designers have as to what the prototypes and designs mean. UI designers, therefore, now have a precise model which makes it easier to communicate with other members of the system development team (which may include not just formal specialists but also other designers) in an unambiguous manner.

From the formal practitioner's point of view, the generation of the models means that it is now possible to interpret UI designs formally and have the same understanding as the designers as to their meaning. They can also make the link between the UI and the system specification (by generating the *PMR*). Just as designers use their informal artefacts as a means of commu-

nicating with users, the formal specialists can use the models as a means of communicating with designers. There is now an agreed meaning to correctness (via presentation models, PIMs and *PMR's*) and an agreed process of refinement (based on contractual utility) which have both a theoretical basis and a light-weight method of use. The formal models and refinement theory allow the UI to be given the same formal basis as the rest of the system.

8.2 Discussion

The process of conducting this research led to the discovery of some congruences between theories which at first appeared surprising. The development of the notion of contractual utility for UI refinement was based on three major components: refinement theory in general; the development process for UIs which led to the understanding of what it might mean to refine a UI; the nature of the models we had developed earlier in our research to support the incorporation of design artefacts into a formal process. Later, when we came to examine this in light of μChart trace refinement we did not expect there to be such a strong relationship between the two.

In particular, the monotonicity results and the link to the validity requirements seemed too unlikely at first. It was only as we unravelled the monotonicity side-conditions that it became clear why the two were so strongly connected. These particular results and the research which led to their understanding made this part of our work particularly exciting and enjoyable.

We have stated in chapter 2 of this thesis that our aim was to provide a general approach not tied to one particular formalism, and that our use of the language Z for the formal specifications was chosen for convenience rather than as a deliberate decision to provide a method for Z users. However, as we have gone on to consider ($UI \parallel Sys$) and refinement, we have pinned down the theory in one particular language, namely μCharts. While we do believe that there are other languages which may be used to formalise refinement, our intention here was to show that such a theory exists and can be used in the manner shown and as such we do not assume a similar generalisation as we have discussed for the early stages of specification and design.

The link between μCharts and Z is convenient for our research, but development of μcharts for both UIs and systems is not reliant on the use of Z earlier in the specification stage, only that the system can be given a general meaning as a reactive system.

The basis of our research was from the viewpoint that UI designers and formal methods practitioners are distinct groups within software development teams; this may not always be the case. Within smaller teams in particular, there may be no such distinction but rather everyone may be involved in all parts of the design and development process. The benefits of our methods in such an environment are of course equally evident in that they enable an easier integration of the different parts of the system and provide the ability to give all parts the same considerations of formality and correctness.

Similarly, for UI designers who are not required to incorporate their work

into a formal setting there are still benefits from using the models. They enable the designers to precisely express their intentions and can be used to consider usability properties, as we have shown. As such, the benefits of our work go beyond our original intentions.

8.3 Future Work

The research presented here is complete in that it provides methods of integration between UI design and formal system development which can be used as soon as initial designs have been created (prototypes or specifications) right through to implementation. However the work has suggested a number of different areas of future work.

The development of tools to support the methods described would be a useful addition. While the PIMed tool has been specified and the UI designed and modelled, the software itself has not yet been implemented, this would be a logical, and useful progression and is currently being undertaken. Once this has been completed is provides the possibility of usability testing of both the tool itself, and the methods it supports, by working with UI designers and web-designers and performing empirical studies. In addition, given that tools for editing and translating μcharts already exist (AMuZed and ZooM [6]) providing a link between these tools and the PIMed application would also be beneficial.

We have deliberately avoided considering the user within our research other

than as an abstract entity with the ability to interact with a system via a UI. There are, however, areas within our work where the consideration of usability (and developing usable systems in general) overlaps with research which does concern itself with user behaviour. As such there is a possibility of extending our research in this direction to include issues of user behaviour and cognition. In particular we are interested in user models, such as PUMs [108] which have been used extensively by Blandford *et al.* [22], [11], [12] in approaches to integrating user models with design models, as well as for usability evaluation at early stages of the design process. It may be interesting to integrate the semi-formal PUMs with presentation models and PIMs as a way of examining the correspondence between expected user behaviours (from the PUM) and behavioural possibilities of the UI (from the presentation model and PIM) leading to a more thorough investigation of early design options.

The approach we have taken to use μCharts as the underlying theory for refinement is only one possible way to consider this problem. A different approach which may prove both beneficial and interesting is to consider UI refinement in the manner of the work of Reeves and Streader [81], [82] which presents a general refinement approach which can be specialised to existing refinement theories. This work considers refinement in terms of contexts and observations and it would be interesting to investigate different interaction requirements (which would allow a single approach to be taken for both horizontal and vertical refinement) as contexts in which a UI design can be placed and define UI refinement from this perspective.

We have introduced vertical refinement as one approach to developing multiple user interfaces for a single application for different platforms but there is scope for more work in this area. Building multiple, or adaptive interfaces from a single design (often referred to as multi-path development) is an important ongoing area of research due to the increasing availability of software applications on different devices and there are different possibilities of using both the models and the vertical refinement approach to support this.

8.4 Final Conclusion

The ability to consider UI designs with a formal system development process provides a number of benefits. We can ensure consistency within the overall design of the system by considering both UI and system by way of formal descriptions. We can also treat the UI design with the same rigour as we do the underlying system without affecting the user-centred and intuitive approach taken by UI designers.

Finding new ways to model design artefacts benefits the UI design process irrespective of whether of not any integration into a formal process is intended. The act of viewing something in another way (by creating a formal model for example) can reveal previously hidden problems or elicit new information. In addition, by examining the embedded knowledge within the designs and making it explicit and formal, as we do in presentation models and PIMs, gives us a new artefact which is unambiguous and can be considered out of its

original context where designer's knowledge is also required for interpretation.

Understanding what it means to refine is a UI and how UI designers currently perform refinement is just as important as finding a way to characterise, and subsequently formalise it. Intuitive design and refinement should not be dismissed because they are informal, but rather we consider them as successful strategies which embody more than just behavioural change. As such our aim is to support and enhance this approach. As with all of our methods refinement by way of contractual utility is intended to be used in conjunction with existing strategies.

Our work can be used in its entirety to support a software development process which is fully formal and which also uses UCD techniques to develop the UI. This was our original intention and we are satisfied we have solved the initial problems we have identified in chapter 1 and met our original goals. We also believe that beyond that our methods can also be used partially to enhance different parts of the design process. For example it is possible to use just the refinement techniques or to compose the system and UI into a valid application, or even use just presentation models and PIMs, and each of these by themselves can add value to the UI and system development processes. What we have then is a technique consisting of the development of models and integration techniques which both satisfies the intention of integrating formal methods and informal UI design as well as providing useful methods and enhancements to existing processes.

Appendix A

Summary of Survey

Overview of Results

A survey of software developers was carried out between March and July 2006. The purpose of the study was to gather information from software developers in New Zealand to try and identify the common practices and processes used in general software design and user interface design. In particular, it aimed to examine the uses of, and attitudes to, formal approaches to software design (especially the use of formal methods) and user-centred design approaches to user interface design.

Two different groups of software developers were targeted. The first group was students enrolled in a post-graduate degree in a New Zealand university who are currently involved in designing software as part of their studies. The second group was people working in professional software development environments in New Zealand. These two different groups were targeted in order to see what, if any, differences exist between software developers in industry and those developing for research purposes. In addition, as the post-graduate students were more likely to have recently been exposed to both formal methods and user-centred design techniques as part of their undergraduate studies, we were interested to see if this had any effect on their approaches to software design and development. Thirdly we considered the two groups to represent

current software developers and future software developers and so were interested in what differences there may be between the two.

The survey consisted of an online questionnaire with two different versions available, one for each of the target participant groups. Student participants were contacted via University Computer Science departments and invited to respond. Professional developers were targeted via information on the usability professional website, invitation to participate included in a usability professional mail out, specific targeting of large companies in New Zealand involved in software development, and postings to IT-based forums on NZ websites.

The survey ran online for five months. Student participant responses were gathered in the first two months and the majority of professional participants in the final three months. Ethical consent for the study was granted by the University of Waikato Ethics Committee in March 2006 and the research participant's bill of rights provided online along with the questionnaires.

Software Developers Questions

General Questions

The following questions will provide us with some background information which will allow us to categorise our respondents into different groups.

- What is your job title?
- Describe your overall job responsibilities

- What size is your organisation?
  - Sole Trader
  - Small
  - Medium
  - Large
- Where is your organisation based?
- Is this organisation solely concerned with software development?
- If no, what is the main activity of your organisation?
- If yes, which areas do you develop software for (choose all that apply)
  - Education
  - Small screen device (e.g. PDA, Mobile Phone)
  - Web based
  - Healthcare
  - Finance
  - Business
  - Gaming
  - Entertainment
  - Safety-Critical
  - Database

  - Other - please specify

- How long has the organisation been established?

  - Less than 1 Year
  - 1-5 Years
  - 5-10 Years
  - More than 10 years

Design Questions

The following questions relate to the software system(s) you design/build. If the answer depends on the particular project then please provide the *average* figure.

- How many people are involved in each development project?
- Of these what percentage are involved with each of the following tasks (please enter a percentage between 1 and 100)

  - Project Management
  - Requirements gathering
  - Formal specification
  - GUI design
  - Implementation/coding
  - Testing

  - User testing
  - Documentation
  - Training
  - Other - please specify

- What is the average time-frame for your projects?
- Of this time, what percentage is spent on the following tasks (please enter a percentage between 0 and 100)
  - Gathering requirements for the functionality of the system
  - Gathering requirements of the users of the system
  - Formally specifying the functionality of the system (including verification and validation of that specification)
  - Informally planning the functionality of the system
  - Designing the GUI to the system
  - Implementing your designs
  - Testing the functionality of the system
  - Usability testing of the system
  - Redesign and re-implementation
  - Preparing documentation
  - Training users
  - Other - please specify

Interface Design Questions

The following questions relate solely to the design/building of the GUI for your software system. Please provide as much information as you can.

- During the design stages is the GUI considered separately from the rest of the system? (*i.e.* as a separate entity)
  - Yes
  - No
- Can you describe in a short paragraph how you go about designing and building the GUI?
- Can you list any things that are created to help with the design or as partial designs along the way?
- Can you list any tools that are used to assist with the methods you have described in your previous two answers?

Implementation Questions

The following questions relate to the implementation of your system, *i.e.* writing the code.

- How far into the overall project does the process of writing code begin? Please select one:
  - At the beginning

  - During the first quarter of the project
  - During the first half of the project
  - During the first three-quarters of the project
  - During the final quarter of the project

- Describe all of the resources used to support the process of writing the code (*e.g.* the things you have described in your previous answers for the design stages of the software plus anything else you may use such as pre-existing items like design guidelines or domain information.)

- Please list any tools used to assist during the coding stage (include IDEs if used)

- Are both the system functionality and the GUI worked on at the same time?
  - Yes
  - No

- If no, at what stage are the two things integrated?
  - When both parts are about a quarter or less complete
  - When both parts are about a half complete
  - When both parts are about three-quarters complete
  - When both parts are complete
  - Will complete one first (which?) and then start on the other.

- Once the coding is complete please describe what, if any, methods of testing are employed

Final Questions

- Do you have any further comments you wish to make regarding the methods, tools and resources used when designing software systems?

- Do you have any further comments you wish to make regarding the methods, tools and resources used when implementing software system?

- Do you have any comments you wish to make regarding this survey?

Summary of Professional Developer Results

Number of participants: 24

Organisation Location:

Auckland	25%
Wellington	42%
Hamilton	4%
Christchurch	25%
Tauranga	4%

Length Organisation Established:

1-5 Years	29%
5-10 Years	17%
>10 Years	54%

Main Core Business Activities:

Software Development	50%
Research	12%
Education	8%
GPS Development	8%
Other	22%

Hardware Developing For:

Desktop	71%
Web Based	67%
Client/Server	42%
Small Screen Device	42%

Domain Developing For:

Education	21%
Business	17%
Entertainment	12%
Finance	8%

Health	8%

Main Activities of Respondents:

Writing code/Implementation	54%
Project Management	17%
Design	17%

Average Development Team Size:

4 - 10

Number of Development Team Members Involved in Task:

1: Writing code

2: Functionality testing

3: Requirements gathering

4: Documentation

5: User testing

6: GUI development

7: Project management

8: Formal specification

9: Training users

10. Other (production support)

(1. being the greatest amount, 10. the least amount)

Typical timeframe for development is less than one year (83%)

Development time spent:

1: Writing code/implementation

2: Functional testing

3: Redesign and reimplementation

4: GUI development

5: Requirements gathering

6: Usability testing

7: Informal planning

8: Formal specification

9: Documentation

10: Training users

(1. being the greatest time, 10. the least amount)

GUI development:

Most common approaches/methods used:

Paper-based prototyping	66%
User-centred approach	80%

Most common development tools used:

Visual Studio	49%

Writing system code:

21% start writing code during first quarter of development time.

Code Writing / Implementation:

Most common tools/aids used:

UML	15%
Sequence diagrams	15%

Development tools used:

Eclipse	29%
Visual Studio	46%

Testing methods used:

Usability testing	42%
Functional testing	25%
Unit testing	17%

Student Developer Questions

General Information Questions

The following questions will provide us with some background information which will allow us to categorise our respondents into different groups.

- What qualification are you currently enrolled in?
- How long have you been studying for this qualification?
- Have you taken any computer science undergraduate course which teaches user centred design methods or HCI principles and practices?
- If yes, how long ago?
  - Less than 1 year
  - 1-2 years ago
  - 2-3 years ago
  - More than 3 years ago
- Have you taken any computer science undergraduate course which teaches software engineering or software design specification techniques and languages?
- If yes, how long ago?
  - Less than 1 year
  - 1-2 years ago
  - 2-3 years ago
  - More than 3 years ago
- Which of the following software specification languages and notations are you familiar with? (Select **all** that apply)

  - Z
  - UML
  - VDM
  - B
  - CSP
  - StateCharts
  - Other - please specify

- Are you currently involved in designing and/or building software as part of your studies? (If no, thank you for your time you are not required to fill in any further information)

Design Questions

The following questions relate to the software system(s) you are designing/building as part of your studies.

- What category does the software you are designing/building fall into (select **all** that apply)
  - Education
  - Small screen device (e.g. PDA, Mobile Phone)
  - Web based
  - Healthcare

  - Finance
  - Business
  - Gaming
  - Entertainment
  - Safety-Critical
  - Database
  - Other - please specify

- Are you solely responsible for designing/building this software?
  - Yes
  - No

- If no, how many other people are involved in this project?

- What is your time-frame for designing/building this software?

- Of this time, what percentage has been/do you anticipate will be spent on the following tasks (please enter a percentage between 0 and 100)
  - Gathering requirements for the functionality of the system
  - Gathering requirements of the users of the system
  - Formally specifying the functionality of the system (including verification and validation of that specification)
  - Informally planning the functionality of the system

  - Designing the GUI to the system
  - Implementing your designs
  - Testing the functionality of the system
  - Usability testing of the system
  - Redesign and re-implementation
  - Preparing documentation
  - Training users
  - Other - please specify

GUI Design Questions

The following questions relate solely to the design/building of the GUI for your software system. Please provide as much information as you can.

- During the design stages for your software did you/will you consider the GUI separately from the rest of the system (*i.e.* as a separate entity)
  - Yes
  - No
- Can you describe in a short paragraph how you went about/will go about designing and building the GUI?
- Can you list any things you will use or create to help with the design or as partial designs along the way?

- Can you list any tools you will use to assist with the methods you have described in your previous two answers?

Student Participant Implementation Questions

The following questions relate to the implementation of your system, *i.e.* writing the code.

- How far into the overall project will you begin writing code? Please select one
  - At the beginning
  - During the first quarter of the project
  - During the first half of the project
  - During the first three-quarters of the project
  - During the final quarter of the project
- Describe all of the resources you will use to support you as you write the code (*e.g.*the things you have described in your previous answers for the design stages of the software plus anything else you may use such as pre-existing items like design guidelines or domain information.)
- Please list any tools you will use to assist during the coding stage (include IDEs if used)
- Will you work on the both the system functionality and the GUI at the same time?

  - Yes
  - No

- If no, at what stage will you integrate the two?
  - When both parts are about a quarter or less complete
  - When both parts are about a half complete
  - When both parts are about three-quarters complete
  - When both parts are complete
  - Will complete one first (which?) and then start on the other.

- Once the coding is complete please describe what, if any, methods of testing you will employ.

Final Questions

- Do you have any further comments you wish to make regarding the methods, tools and resources you use when designing your software system?

- Do you have any further comments you wish to make regarding the methods, tools and resources you use when implementing your software system?

- Do you have any comments you wish to make regarding this survey?

Summary of Student Results

Number of participants:	38
PhD Candidates	47%
Masters Students	32%
Other	21%

Length of current studies:

$<$ 2 years	68%
$>$ 2 years	32%

39% have taken HCI course (13% within last 12 months)

68% have taken a course on formal methods/specifications

94% are familiar with at least one method of specification

- 74% familiar with UML
- 18% familiar with Z

82% are currently developing software

15% develop for multiple platforms

- 61% for desktop machines
- 11% for small screen devices

15% develop for multiple domains

34% develop business applications

32% develop education applications

24% develop entertainment applications

79% develop alone

Typical timeframe for development is less than one year (69%)

Development time spent:

1: Writing code

2: Redesign and reimplementation

3: Functional testing

4: GUI development

5: Informal planning

6: Usability testing

7: Requirements gathering

8: Documentation

9: Formal specification

10: Training users (zero time spent by any respondents)

(1. being the greatest time, 10. the least amount)

GUI Development:

Most common methods and visual aids used:

Sketching of designs and prototypes	50%
Prototyping with users	22%
Interactive whiteboard design sessions	5%

Most common development tools used:

Eclipse	22%
Java/Swing	17%
Dreamweaver	11%

Writing System Code:

42% start writing code during first quarter of development time

Code Writing / Implementation:

Most common tools/aids used:

UML class diagrams	21%
Xtreme programming techniques	7%

Most common development tools/IDEs used:

Eclipse	50%
Visual Studio	10%

Testing methods used:

User and usability testing	21%

Unit testing	13%
Functional testing	8%

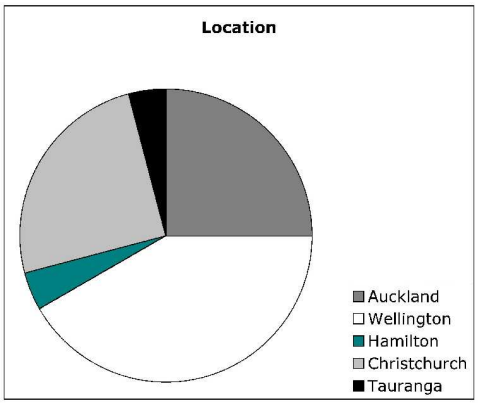


Figure 8.2: Locations

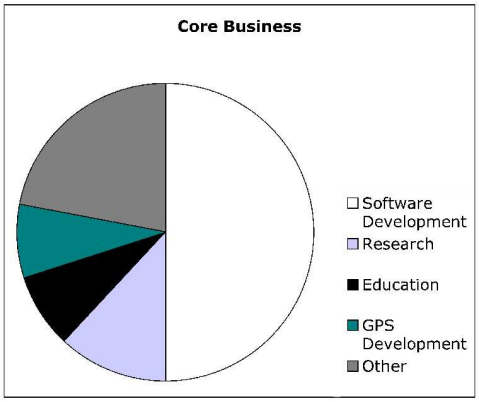


Figure 8.3: Core Business

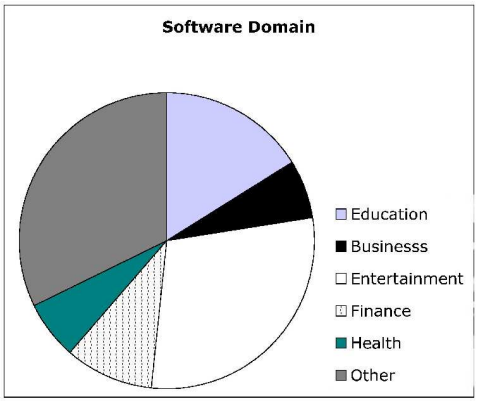


Figure 8.4: Software Domain

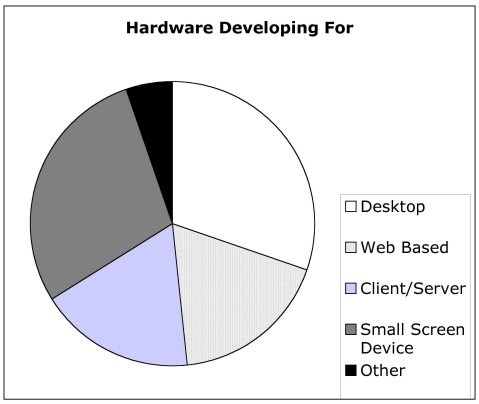


Figure 8.5: Hardware

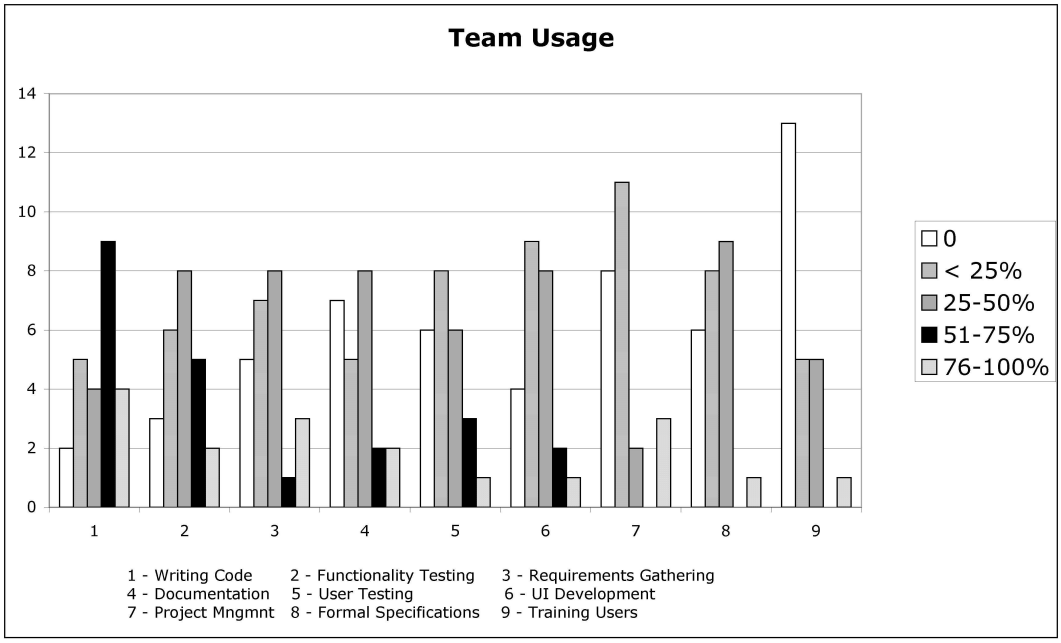


Figure 8.6: Team Usage

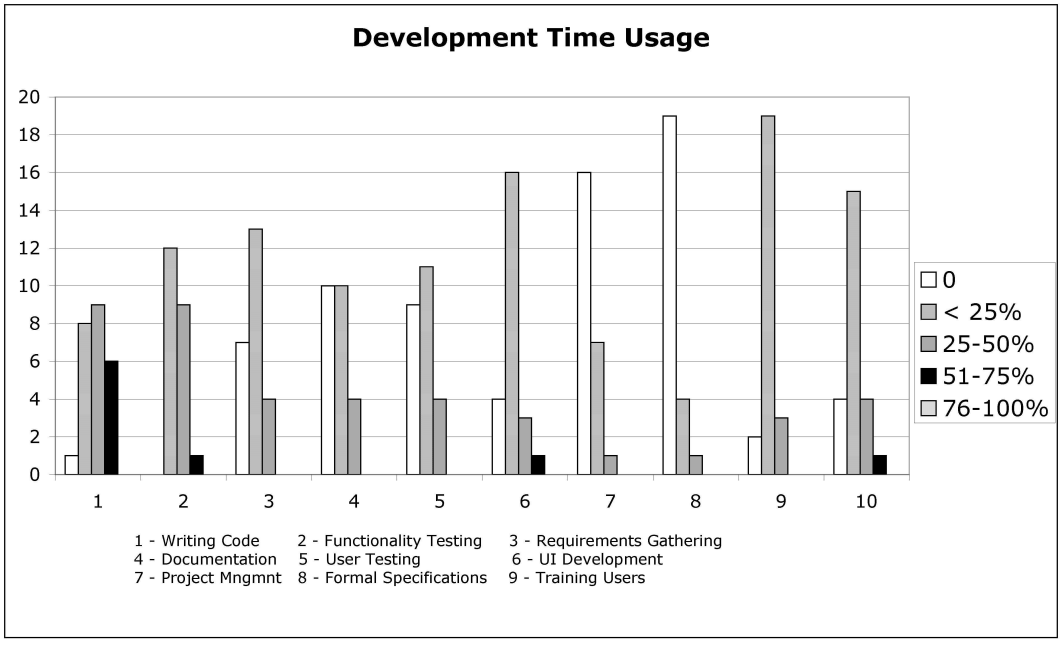
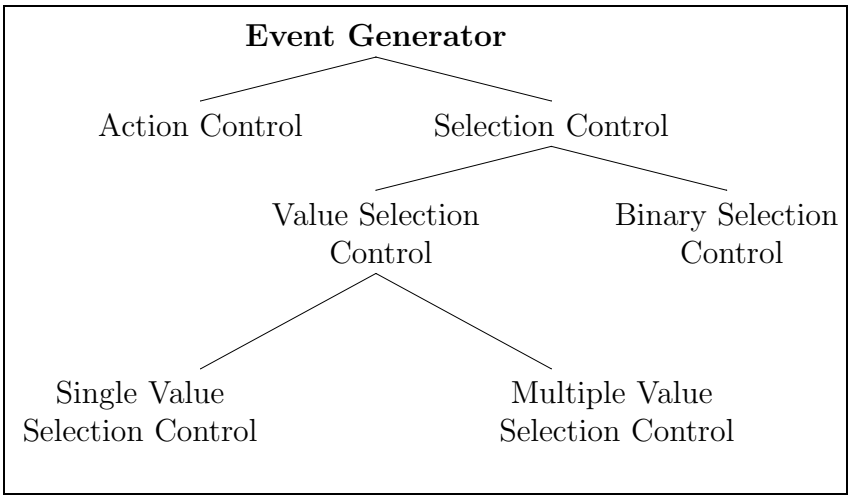


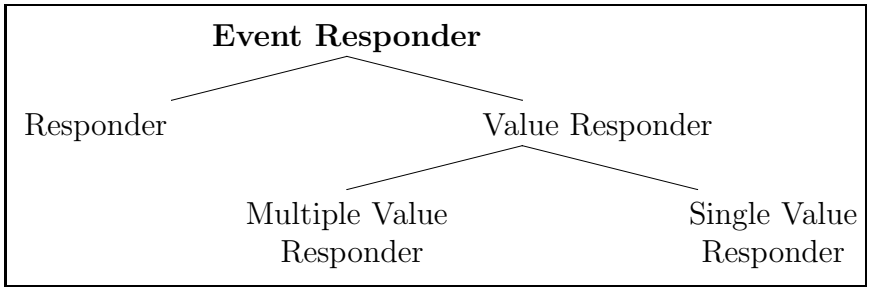
Figure 8.7: Development Time Usage

Appendix B

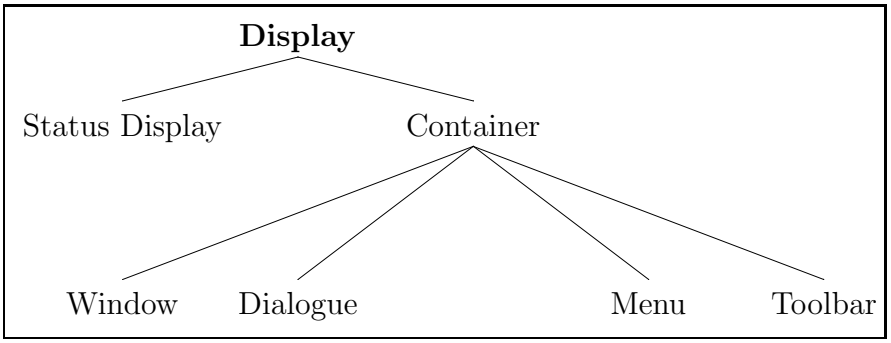
Widget Category Hierarchies



Event Generator Hierarchy



Event Responder Hierarchy



Display Hierarchy

Appendix C

Presentation Model Semantics Example

Given the following presentation model we show how the semantic functions defined in chapter 4 build the correct environment.

PModel	MainApp WinA WinB
WidgetName	ControlOne SelTwo ControlThree
Category	ActCtrl SValSelector
Behaviour	DoActionA DoActionB DoActionC
WinA is	(ControlOne,ActCtrl,(DoActionA))
	(SelTwo, SValSelector,(DoActionB))
WinB is	(ControlThree, ActCtrl,(DoActionC))

MainApp is WinA : WinB

As an abbreviation we call the declarations (the first four lines of the presentation model above) *Decl* and the definitions (the last four lines) *Def*. We create

the environment using the semantic functions defined in chapter 4.

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

$$\Rightarrow$$

$$Df[\![Def]\!](Dc[\![PModel\ MainApp\ WinA\ WinB\ \ WidgetName\ ControlOne\ SelTwo\ ControlThree\ \ Category\ ActCtrl\ SValSelector\ \ Behaviour\ DoActionA\ DoActionB\ DoActionC]\!])$$

$$\Rightarrow$$

$$Df[\![Def]\!](\{MainApp \mapsto \overline{MainApp}, WinA \mapsto \overline{WinA}, WinB \mapsto \overline{WInB}\} \cup \{ControlOne \mapsto \overline{ControlOne}, SelTwo \mapsto \overline{SelTwo}, ControlThree \mapsto \overline{ControlThree}\} \cup \{ActCtrl \mapsto \overline{ActCtrl}, SValSelector \mapsto \overline{SValSelector}\} \cup \{DoActionA \mapsto \overline{DoActionA}, DoActionB \mapsto \overline{DoActionB}, DoActionC \mapsto \overline{DoActionC}\})$$

$$\Rightarrow$$

$$Df[\![Def]\!](\{MainApp \mapsto \overline{MainApp}, WinA \mapsto \overline{WinA}, WinB \mapsto \overline{WInB}, ControlOne \mapsto \overline{ControlOne}, SelTwo \mapsto \overline{SelTwo}, ControlThree \mapsto \overline{ControlThree}, ActCtrl \mapsto \overline{ActCtrl}, SValSelector \mapsto \overline{SValSelector}, DoActionA \mapsto \overline{DoActionA}, DoActionB \mapsto \overline{DoActionB}, DoActionC \mapsto \overline{DoActionC}\})$$

Now the ϵ we use in the definitions section refers to the populated environment we have given above.

$$Df[\![D\ Ds]\!]\epsilon = Df[\![Ds]\!](Df[\![D]\!]\epsilon)$$

$$\Rightarrow$$

$$Df[\![Ds]\!](Df[\![\mathit{WinA}\ \mathit{is}\ (\mathit{ControlOne}, \mathit{ActCtrl}, (\mathit{DoActionOne})),$$

$$(\mathit{SelTwo}, \mathit{SValSelector}, (\mathit{DoActionB}))]\!]\epsilon)$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon \oplus \{\mathit{WinA} \mapsto \mathit{Expr}[\![(\mathit{ControlOne}, \mathit{ActCtrl}, (\mathit{DoActionOne})),$$

$$(\mathit{SelTwo}, \mathit{SValSelector}, (\mathit{DoActionB}))]\!]\epsilon\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon \oplus \{\mathit{WinA} \mapsto \mathit{Expr}[\![(\mathit{ControlOne}, \mathit{ActCtrl}, (\mathit{DoActionOne}))]\!]\epsilon$$

$$\cup\, \mathit{Expr}[\![Es]\!]\epsilon\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon \oplus \{\mathit{WinA} \mapsto \{((\overline{\mathit{ControlOne}})\ (\overline{\mathit{ActCtrl}})\ \{(\overline{\mathit{ActionA}})\})\}$$

$$\cup\, \mathit{Expr}[\![Es]\!]\epsilon\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon \oplus \{\, WinA \mapsto \{((\overline{ControlOne})\,(\overline{ActCtrl})\,\{(\overline{ActionA})\})\}$$

$$\cup\, Expr[\![(SelTwo, SValSelector, (ActionB))]\!]\epsilon\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon \oplus \{\, WinA \mapsto \{((\overline{ControlOne})\,(\overline{ActCtrl})\,\{(\overline{ActionA})\})\}$$

$$\cup \{((\overline{SelTwo})\,(\overline{SValSelector})\,\{(\overline{ActionB})\})\}\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon \oplus \{\, WinA \mapsto \{((\overline{ControlOne})\,(\overline{ActCtrl})\,\{(\overline{ActionA})\}),$$

$$((\overline{SelTwo})\,(\overline{SValSelector})\,\{(\overline{ActionB})\})\}\})$$

$$Let\, \epsilon' = \epsilon \oplus \{\, WinA \mapsto \{((\overline{ControlOne})\,(\overline{ActCtrl})\,\{(\overline{ActionA})\}),$$

$$((\overline{SelTwo})\,(\overline{SValSelector})\,\{(\overline{ActionB})\})\}\}$$

$$\Rightarrow$$

$$Df[\![Ds]\!](Df[\![\, WinB\ is\ (ControlThree, ActCtrl, (DoActionC))]\!]\epsilon')$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon' \oplus \{\, WinB \mapsto Expr[\![(ControlThree, ActCtrl, (DoActionC))]\!]\epsilon'\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon' \oplus \{\, WinB \mapsto \{((\overline{ControlThree})\,(\overline{ActCtrl})\,\{(\overline{DoActionC})\})\})$$

$$Let\ \epsilon'' = \epsilon' \oplus \{WinB \mapsto \{((\overline{ControlThree})\ (\overline{ActCtrl})\ \{(\overline{DoActionC})\})\}$$

$$\Rightarrow$$

$$Df[\![Ds]\!](Df[\![MainApp\ is\ WinA\ :\ WinB]\!]\epsilon'')$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon'' \oplus \{MainApp \mapsto Expr[\![\ WinA : WinB]\!]\epsilon''\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon'' \oplus \{MainApp \mapsto Expr[\![\ WinA]\!]\epsilon'' \cup Expr[\![\ WinB]\!]\epsilon''\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon'' \oplus \{MainApp \mapsto \{((\overline{ControlOne})\ (\overline{ActCtrl})\ \{(\overline{DoActionA})\}),$$
$$((\overline{SelTwo})\ (\overline{SValSelector})\ \{(\overline{DoActionB})\})\} \cup$$
$$\{((\overline{ControlThree})\ (\overline{ActCtrl})\ \{(\overline{DoActionC})\})\}\})$$

$$\Rightarrow$$

$$Df[\![Ds]\!](\epsilon'' \oplus \{MainApp \mapsto \{((\overline{ControlOne})\ (\overline{ActCtrl})\ \{(\overline{DoActionA})\}),$$
$$((\overline{SelTwo})\ (\overline{SValSelector})\ \{(\overline{DoActionB})\}),$$
$$((\overline{ControlThree})\ (\overline{ActCtrl})\ \{(\overline{DoActionC})\})\}\})$$

$$Let\ \epsilon''' = \epsilon'' \oplus \{MainApp \mapsto \{((\overline{ControlOne})\ (\overline{ActCtrl})\ \{(\overline{DoActionA})\}),$$
$$((\overline{SelTwo})\ (\overline{SValSelector})\ \{(\overline{DoActionB})\}),$$
$$((\overline{ControlThree})\ (\overline{ActCtrl})\ \{(\overline{DoActionC})\})\}\}$$

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

$\{ ControlOne \mapsto \overline{ControlOne}, SelTwo \mapsto \overline{SelTwo}, ControlThree \mapsto$

$\overline{ControlThree}, ActCtrl \mapsto \overline{ActCtrl}, SValSelector \mapsto \overline{SValSelector},$

$DoActionA \mapsto \overline{DoActionA}, DoActionB \mapsto \overline{DoActionB},$

$DoActionC \mapsto \overline{DoActionC}, WinA \mapsto \{((\overline{ControlOne})(\overline{ActCtrl})$

$\{(\overline{DoActionA}), ((\overline{SelTwo})\ (\overline{SValSelector})\ \{(\overline{DoActionB})\})\},$

$WinB \mapsto \{((\overline{ControlThree})\ (\overline{ActCtrl})\ \{(\overline{DoActionC})\})\},$

$MainApp \mapsto \{((\overline{ControlOne})(\overline{ActCtrl})\ \{(\overline{DoActionA})\}),$

$((\overline{SelTwo})\ (\overline{SValSelector})\ \{(\overline{DoActionB})\}), ((\overline{ControlThree})$

$(\overline{ActCtrl})\ \{(\overline{DoActionC})\})\}$

Appendix D

The following Z description was auto-generated by the ZooM tool [6] from output provided by the AMuZed tool [6].

$$\mu_{State} ::= ShapeApp \mid ShapeAppOpenWin \mid ShapeAppCircleWin$$
$$\mid ShapeAppSquareWin \mid ShapeAppTriangleWin$$

$$Signal ::= SICirclewin \mid SITriangleWin \mid SISquareWin \mid SICircleWin$$

$$states_{ShapeApp} : \mathbb{P}\, \mu_{State}$$

$$inputI_{ShapeApp} : \mathbb{P}\, Signal$$

$$outputI_{ShapeApp} : \mathbb{P}\, Signal$$

$$\Psi_{ShapeApp} : \mathbb{P}\, Signal$$

$$states_{ShapeApp} = \{ShapeAppOpenWin, ShapeAppCircleWin,$$
$$ShapeAppSquareWin, ShapeAppTriangleWin\}$$

$$inputI_{ShapeApp} = \{SICirclewin, SICircleWin, SITriangleWin,$$
$$SISquareWin\}$$

$$outputI_{ShapeApp} = \{\}$$

$$\Psi_{ShapeApp} = \{\}$$

$Chart_{ShapeApp}$

$c_{ShapeApp} : states_{ShapeApp}$

$Init_{ShapeApp}$

$Chart_{ShapeApp}$

$c_{ShapeApp} = ShapeAppOpenWin$

$ShapeAppOpenWin$

$Chart_{ShapeApp}$

$c_{ShapeApp} = ShapeAppOpenWin$

$ShapeAppCircleWin$

$Chart_{ShapeApp}$

$c_{ShapeApp} = ShapeAppCircleWin$

ShapeAppSquareWin

$Chart_{ShapeApp}$

$c_{ShapeApp} = ShapeAppSquareWin$

ShapeAppTriangleWin

$Chart_{ShapeApp}$

$c_{ShapeApp} = ShapeAppTriangleWin$

$\delta_{OpenWinCircleWin}$

$ShapeAppOpenWin$

$ShapeAppCircleWin'$

$i_{ShapeApp}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$active(ShapeApp)$

$SICircleWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{OpenWinSquareWin}$

$ShapeAppOpenWin$

$ShapeAppSquareWin'$

$i_{ShapeApp}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$active(ShapeApp)$

$SISquareWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{OpenWinTriangleWin}$

$ShapeAppOpenWin$

$ShapeAppTriangleWin'$

$i_{ShapeApp}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$active(ShapeApp)$

$SITriangleWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{SquareWinCircleWin}$

$ShapeAppSquareWin$

$ShapeAppCircleWin'$

$i_{ShapeApp}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$active(ShapeApp)$

$SICirclewin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{CircleWinTriangleWin}$

$ShapeAppCircleWin$

$ShapeAppTriangleWin'$

$i_{ShapeApp}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$active(ShapeApp)$

$SITriangleWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{CircleWinSquareWin}$

$ShapeAppCircleWin$

$ShapeAppSquareWin'$

$i_{ShapeApp}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$active(ShapeApp)$

$SISquareWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{TriangleWinCircleWin}$

$ShapeAppTriangleWin$

$ShapeAppCircleWin'$

$i_{ShapeApp}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$active(ShapeApp)$

$SICircleWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{SquareWinTriangleWin}$

$ShapeAppSquareWin$

$ShapeAppTriangleWin'$

$i_{ShapeApp}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\ outputI_{ShapeApp}$

$active(ShapeApp)$

$SITriangleWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$\delta_{TriangleWinSquareWin}$

$ShapeAppTriangleWin$

$ShapeAppSquareWin'$

$i_{ShapeApp}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\ outputI_{ShapeApp}$

$active(ShapeApp)$

$SISquareWin \in i_{ShapeApp}? \cup (o_{ShapeApp}! \cap \Psi_{ShapeApp})$

$o_{ShapeApp}! = \{\}$

$Iactive_{ShapeApp}$

$\Xi Chart_{ShapeApp}$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$\neg\ active(ShapeApp)$

$o_{ShapeApp}! = \{\}$

$$\delta_{ShapeApp} \mathrel{\widehat{=}} \delta_{OpenWinCircleWin} \lor \delta_{OpenWinSquareWin} \lor \delta_{OpenWinTriangleWin}$$
$$\lor \delta_{SquareWinCircleWin} \lor \delta_{CircleWinTriangleWin} \lor \delta_{CircleWinSquareWin}$$
$$\lor \delta_{TriangleWinCircleWin} \lor \delta_{SquareWinTriangleWin} \lor \delta_{TriangleWinSquareWin}$$
$$\lor\ Iactive_{ShapeApp}$$

$Init_{ShapeAppSys}$

$Init_{ShapeApp}$

ShapeAppSys

$\Delta Chart_{ShapeApp}$

$i_{ShapeApp}? : \mathbb{P}\, inputI_{ShapeApp}$

$o_{ShapeApp}! : \mathbb{P}\, outputI_{ShapeApp}$

$\exists\, active\_ : \mathbb{P}\, \mu_{State} \bullet$

$\quad active(ShapeApp) \land \delta_{ShapeApp}$

Appendix E

The following Z description was auto-generated by the ZooM tool [6] from output provided by the AMuZed tool [6].

$$\mu_{State} ::= ShapeUI \mid ShapeSystem \mid ShapeUIShapeSystem \mid ShapeUISystemR$$
$$\mid ShapeUIOpenWin \mid ShapeUICircleWin \mid ShapeUISquareWin$$
$$\mid ShapeUITriangleWin \mid ShapeSystemInit \mid ShapeSystemShapeIsCircle$$
$$\mid ShapeSystemShapeIsSquare \mid ShapeSystemShapeIsTriangle$$

$$Signal ::= SSShowTriangle \mid SITriangleWin \mid SSShowSquare \mid SISquareWin$$
$$\mid SSShowCircle \mid SICircleWin \mid SSelectCircle \mid SSelectSquare \mid$$
$$SSelectTriangle$$

$states_{ShapeUI} : \mathbb{P}\, \mu_{State}$

$inputI_{ShapeUI} : \mathbb{P}\, Signal$

$outputI_{ShapeUI} : \mathbb{P}\, Signal$

$\Psi_{ShapeUI} : \mathbb{P}\, Signal$

$states_{ShapeUI} = \{ShapeUIOpenWin, ShapeUICircleWin,$

$\quad ShapeUISquareWin, ShapeUITriangleWin\}$

$inputI_{ShapeUI} = \{SSShowTriangle, SSShowSquare, SSShowCircle,$

$\quad SISquareWin, SICircleWin, SITriangleWin\}$

$outputI_{ShapeUI} = \{SSelectTriangle, SSelectSquare, SSelectCircle\}$

$\Psi_{ShapeUI} = \{\}$

$Chart_{ShapeUI}$

$c_{ShapeUI} : states_{ShapeUI}$

$Init_{ShapeUI}$

$Chart_{ShapeUI}$

$c_{ShapeUI} = ShapeUIOpenWin$

ShapeUIOpenWin

$Chart_{ShapeUI}$

$c_{ShapeUI} = ShapeUIOpenWin$

ShapeUICircleWin

$Chart_{ShapeUI}$

$c_{ShapeUI} = ShapeUICircleWin$

ShapeUISquareWin

$Chart_{ShapeUI}$

$c_{ShapeUI} = ShapeUISquareWin$

ShapeUITriangleWin

$Chart_{ShapeUI}$

$c_{ShapeUI} = ShapeUITriangleWin$

$\delta_{OpenWinCircleWin}$

$ShapeUIOpenWin$

$ShapeUICircleWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SICircleWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\wedge\ SSShowCircle \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectCircle\}$

$\delta_{OpenWinSquareWin}$

$ShapeUIOpenWin$

$ShapeUISquareWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SISquareWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowSquare \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectSquare\}$

$\delta_{SquareWinCircleWin}$

$ShapeUISquareWin$

$ShapeUICircleWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SICircleWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowCircle \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectCircle\}$

$\delta_{OpenWinTriangleWin}$

$ShapeUIOpenWin$

$ShapeUITriangleWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SITriangleWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowTriangle \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectTriangle\}$

$\delta_{SquareWinTriangleWin}$

$ShapeUISquareWin$

$ShapeUITriangleWin'$

$i_{ShapeUI}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\ outputI_{ShapeUI}$

$active(ShapeUI)$

$(SITriangleWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowTriangle \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectTriangle\}$

$\delta_{CircleWinTriangleWin}$

$ShapeUICircleWin$

$ShapeUITriangleWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SITriangleWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowTriangle \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectTriangle\}$

$\delta_{CircleWinSquareWin}$

$ShapeUICircleWin$

$ShapeUISquareWin'$

$i_{ShapeUI}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\ outputI_{ShapeUI}$

$active(ShapeUI)$

$(SISquareWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowSquare \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectSquare\}$

$\delta_{TriangleWinSquareWin}$

$ShapeUITriangleWin$

$ShapeUISquareWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SISquareWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowSquare \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectSquare\}$

$\delta_{TriangleWinCircleWin}$

$ShapeUITriangleWin$

$ShapeUICircleWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SICircleWin \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SSShowCircle \in i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{SSelectCircle\}$

$\delta_{OpenWinOpenWin}$

$ShapeUIOpenWin$

$ShapeUIOpenWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SICircleWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\wedge\ (SISquareWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\wedge\ SITriangleWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})))$

$o_{ShapeUI}! = \{\}$

$\delta_{CircleWinCircleWin}$

$ShapeUICircleWin$

$ShapeUICircleWin'$

$i_{ShapeUI}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\ outputI_{ShapeUI}$

$active(ShapeUI)$

$(SISquareWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SITriangleWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{\}$

$\delta_{TriangleWinTriangleWin}$

$ShapeUITriangleWin$

$ShapeUITriangleWin'$

$i_{ShapeUI}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\, outputI_{ShapeUI}$

$active(ShapeUI)$

$(SICircleWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SISquareWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{\}$

$\delta_{SquareWinSquareWin}$

$ShapeUISquareWin$

$ShapeUISquareWin'$

$i_{ShapeUI}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\ outputI_{ShapeUI}$

$active(ShapeUI)$

$(SICircleWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI})$

$\land\ SITriangleWin \notin i_{ShapeUI}? \cup (o_{ShapeUI}! \cap \Psi_{ShapeUI}))$

$o_{ShapeUI}! = \{\}$

$Iactive_{ShapeUI}$

$\Xi Chart_{ShapeUI}$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeUI}! : \mathbb{P}\ outputI_{ShapeUI}$

$\neg\ active(ShapeUI)$

$o_{ShapeUI}! = \{\}$

$$\delta_{ShapeUI} \mathrel{\widehat{=}} \delta_{OpenWinCircleWin} \lor \delta_{OpenWinSquareWin} \lor \delta_{SquareWinCircleWin}$$
$$\lor \delta_{OpenWinTriangleWin} \lor \delta_{SquareWinTriangleWin} \lor \delta_{CircleWinTriangleWin}$$
$$\lor \delta_{CircleWinSquareWin} \lor \delta_{TriangleWinSquareWin} \lor \delta_{TriangleWinCircleWin}$$
$$\lor \delta_{OpenWinOpenWin} \lor \delta_{CircleWinCircleWin} \lor \delta_{TriangleWinTriangleWin}$$
$$\lor \delta_{SquareWinSquareWin} \lor Iactive_{ShapeUI}$$

$states_{ShapeSystem} : \mathbb{P}\, \mu_{State}$

$inputI_{ShapeSystem} : \mathbb{P}\, Signal$

$outputI_{ShapeSystem} : \mathbb{P}\, Signal$

$\Psi_{ShapeSystem} : \mathbb{P}\, Signal$

$states_{ShapeSystem} = \{ShapeSystemInit, ShapeSystemShapeIsCircle,$
$ShapeSystemShapeIsSquare, ShapeSystemShapeIsTriangle\}$

$inputI_{ShapeSystem} = \{SSelectTriangle, SSelectCircle, SSelectSquare\}$

$outputI_{ShapeSystem} = \{\}$

$\Psi_{ShapeSystem} = \{\}$

$Chart_{ShapeSystem}$

$c_{ShapeSystem} : states_{ShapeSystem}$

$Init_{ShapeSystem}$

$Chart_{ShapeSystem}$

$c_{ShapeSystem} = ShapeSystemInit$

$ShapeSystemInit$

$Chart_{ShapeSystem}$

$c_{ShapeSystem} = ShapeSystemInit$

$ShapeSystemShapeIsCircle$

$Chart_{ShapeSystem}$

$c_{ShapeSystem} = ShapeSystemShapeIsCircle$

$ShapeSystemShapeIsSquare$

$Chart_{ShapeSystem}$

$c_{ShapeSystem} = ShapeSystemShapeIsSquare$

ShapeSystemShapeIsTriangle

$Chart_{ShapeSystem}$

$c_{ShapeSystem} = ShapeSystemShapeIsTriangle$

$\delta_{InitShapeIsCircle}$

$ShapeSystemInit$

$ShapeSystemShapeIsCircle'$

$i_{ShapeSystem}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\, outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectCircle \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{InitShapeIsSquare}$

$ShapeSystemInit$

$ShapeSystemShapeIsSquare'$

$i_{ShapeSystem}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\, outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectSquare \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{InitShapeIsTriangle}$

$ShapeSystemInit$

$ShapeSystemShapeIsTriangle'$

$i_{ShapeSystem}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\, outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectTriangle \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsCircleShapeIsSquare}$

$ShapeSystemShapeIsCircle$

$ShapeSystemShapeIsSquare'$

$i_{ShapeSystem}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\ outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectSquare \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsSquareShapeIsCircle}$

$ShapeSystemShapeIsSquare$

$ShapeSystemShapeIsCircle'$

$i_{ShapeSystem}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\ outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectCircle \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsSquareShapeIsTriangle}$

$ShapeSystemShapeIsSquare$

$ShapeSystemShapeIsTriangle'$

$i_{ShapeSystem}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\, outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectTriangle \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsTriangleShapeIsSquare}$

$ShapeSystemShapeIsTriangle$

$ShapeSystemShapeIsSquare'$

$i_{ShapeSystem}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\, outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectSquare \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsCircleShapeIsTriangle}$

$ShapeSystemShapeIsCircle$

$ShapeSystemShapeIsTriangle'$

$i_{ShapeSystem}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\ outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectTriangle \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsTriangleShapeIsCircle}$

$ShapeSystemShapeIsTriangle$

$ShapeSystemShapeIsCircle'$

$i_{ShapeSystem}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\ outputI_{ShapeSystem}$

$active(ShapeSystem)$

$SSelectCircle \in i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$o_{ShapeSystem}! = \{\}$

$\delta_{InitInit}$

$ShapeSystemInit$

$ShapeSystemInit'$

$i_{ShapeSystem}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\, outputI_{ShapeSystem}$

$active(ShapeSystem)$

$(SSelectCircle \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$\land\ (SSelectSquare \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$\land\ SSelectTriangle \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})))$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsCircleShapeIsCircle}$

$ShapeSystemShapeIsCircle$

$ShapeSystemShapeIsCircle'$

$i_{ShapeSystem}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\ outputI_{ShapeSystem}$

$active(ShapeSystem)$

$(SSelectSquare \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$\land\ SSelectTriangle \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem}))$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsSquareShapeIsSquare}$

$ShapeSystemShapeIsSquare$

$ShapeSystemShapeIsSquare'$

$i_{ShapeSystem}? : \mathbb{P}\, Signal$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\, outputI_{ShapeSystem}$

$active(ShapeSystem)$

$(SSelectCircle \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$\land\ SSelectTriangle \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem}))$

$o_{ShapeSystem}! = \{\}$

$\delta_{ShapeIsTriangleShapeIsTriangle}$

$ShapeSystemShapeIsTriangle$

$ShapeSystemShapeIsTriangle'$

$i_{ShapeSystem}? : \mathbb{P}\ Signal$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\ outputI_{ShapeSystem}$

$active(ShapeSystem)$

$(SSelectCircle \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem})$

$\land\ SSelectSquare \notin i_{ShapeSystem}? \cup (o_{ShapeSystem}! \cap \Psi_{ShapeSystem}))$

$o_{ShapeSystem}! = \{\}$

$Iactive_{ShapeSystem}$

$\Xi Chart_{ShapeSystem}$

$active\_ : \mathbb{P}\ \mu_{State}$

$o_{ShapeSystem}! : \mathbb{P}\ outputI_{ShapeSystem}$

$\neg\ active(ShapeSystem)$

$o_{ShapeSystem}! = \{\}$

$$\delta_{ShapeSystem} \mathrel{\widehat{=}} \delta_{InitShapeIsCircle} \lor \delta_{InitShapeIsSquare} \lor \delta_{InitShapeIsTriangle}$$
$$\lor \delta_{ShapeIsCircleShapeIsSquare} \lor \delta_{ShapeIsSquareShapeIsCircle} \lor \delta_{ShapeIsSquareShapeIsTriangle}$$
$$\lor \delta_{ShapeIsTriangleShapeIsSquare} \lor \delta_{ShapeIsCircleShapeIsTriangle} \lor \delta_{ShapeIsTriangleShapeIsCircle}$$
$$\lor \delta_{InitInit} \lor \delta_{ShapeIsCircleShapeIsCircle} \lor \delta_{ShapeIsSquareShapeIsSquare}$$
$$\lor \delta_{ShapeIsTriangleShapeIsTriangle} \lor Iactive_{ShapeSystem}$$

$states_{ShapeUIShapeSystem} : \mathbb{P}\, \mu_{State}$

$inputI_{ShapeUIShapeSystem} : \mathbb{P}\, Signal$

$outputI_{ShapeUIShapeSystem} : \mathbb{P}\, Signal$

$\Psi_{ShapeUIShapeSystem} : \mathbb{P}\, Signal$

$states_{ShapeUIShapeSystem} = states_{ShapeUI} \cup states_{ShapeSystem}$

$inputI_{ShapeUIShapeSystem} = inputI_{ShapeUI} \cup inputI_{ShapeSystem}$

$outputI_{ShapeUIShapeSystem} = outputI_{ShapeUI} \cup outputI_{ShapeSystem}$

$\Psi_{ShapeUIShapeSystem} = \{SSelectCircle, SSelectSquare, SSelectTriangle\}$

$Chart_{ShapeUIShapeSystem}$

$Chart_{ShapeUI}$

$Chart_{ShapeSystem}$

$Init_{ShapeUIShapeSystem}$

$Init_{ShapeUI}$

$Init_{ShapeSystem}$

$\delta_{ShapeUIShapeSystem}$

$\Delta Chart_{ShapeUIShapeSystem}$

$i_{ShapeUIShapeSystem}? : \mathbb{P}\, inputI_{ShapeUIShapeSystem}$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUIShapeSystem}! : \mathbb{P}\, outputI_{ShapeUIShapeSystem}$

$active(ShapeUI) \Leftrightarrow active(ShapeSystem)$

$\exists\, i_{ShapeUI}?, i_{ShapeSystem}?, o_{ShapeUI}!, o_{ShapeSystem}! : \mathbb{P}\, Signal \bullet$

$i_{ShapeUI}? = (i_{ShapeUIShapeSystem}? \cup (o_{ShapeUIShapeSystem}!$

$\quad \cap \Psi_{ShapeUIShapeSystem}))$

$\quad \cap inputI_{ShapeUI} \land$

$i_{ShapeSystem}? = (i_{ShapeUIShapeSystem}? \cup (o_{ShapeUIShapeSystem}!$

$\quad \cap \Psi_{ShapeUIShapeSystem}))$

$\quad \cap inputI_{ShapeSystem} \land$

$o_{ShapeUIShapeSystem}! = o_{ShapeUI}! \cup o_{ShapeSystem}! \land$

$\delta_{ShapeUI} \land \delta_{ShapeSystem}$

$Chart_{ShapeUISystemR}$

$Chart_{ShapeUIShapeSystem}$

$Init_{ShapeUISystemR}$

$Init_{ShapeUIShapeSystem}$

$states_{ShapeUISystemR} : \mathbb{P}\, \mu_{State}$

$in_{ShapeUISystemR} : \mathbb{P}\, Signal$

$out_{ShapeUISystemR} : \mathbb{P}\, Signal$

$states_{ShapeUISystemR} = states_{ShapeUIShapeSystem}$

$in_{ShapeUISystemR} = \{SICircleWin, SISquareWin, SITriangleWin, SSShowCircle,$

$SSShowSquare, SSShowTriangle\}$

$out_{ShapeUISystemR} = \{\}$

$\delta_{ShapeUISystemR}$

$\Delta Chart_{ShapeUISystemR}$

$i_{ShapeUISystemR}? : \mathbb{P}\, in_{ShapeUISystemR}$

$active\_ : \mathbb{P}\, \mu_{State}$

$o_{ShapeUISystemR}! : \mathbb{P}\, out_{ShapeUISystemR}$

$active(ShapeUISystemR) \Leftrightarrow active(ShapeUIShapeSystem)$

$\exists\, i_{ShapeUIShapeSystem}?, o_{ShapeUIShapeSystem}! : \mathbb{P}\, Signal \bullet$

$i_{ShapeUISystemR}? \cap inputI_{ShapeUIShapeSystem} = i_{ShapeUIShapeSystem}? \wedge$

$o_{ShapeUISystemR}! = o_{ShapeUIShapeSystem}! \cap out_{ShapeUISystemR} \wedge$

$\delta_{ShapeUIShapeSystem}$

$Init_{ShapeUISys}$

$Init_{ShapeUISystemR}$

ShapeUISys

$\Delta Chart_{ShapeUISystemR}$

$i_{ShapeUISystemR}? : \mathbb{P}\, in_{ShapeUISystemR}$

$o_{ShapeUISystemR}! : \mathbb{P}\, out_{ShapeUISystemR}$

$\exists\, active\_ : \mathbb{P}\, \mu_{State} \bullet$

$\delta_{ShapeUISystemR}$

Bibliography

[1] ISO/IEC 13407. *Human-Centred Design Processes for Interactive Systems*. ISO/IEC, first edition, 1999.

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

[3] ISO/IEC 13817-1. *Information Technology—Programming languages, their environments and system software interface—Vienna Development Method—Specification Language—Part 1: Base language*. ISO/IEC, first edition, 1996.

[4] R. Abi-Aad, D. Sinnig, T. Radhakrishnan, and A. Seffah. CoU: Context of use model for user interface designing. In *Proceedings of HCI International 2003, vol.4*, pages 8–12. LEA, 2003.

[5] J. R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[6] AMuZed and ZooM, tools for microcharts. Website with background information and downloads at:

`http://www.cs.waikato.ac.nz/Research/fm/amuzed.html`.

[7] R-J. Back, A. Mikhajlova, and J. von Wright. Reasoning about interactive systems. In *FM '99: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1460–1476, London, UK, 1999. Springer-Verlag.

[8] R. Banach and M. Poppleton. Retrenchment: An engineering variation on refinement. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 129–147, London, UK, 1998. Springer-Verlag.

[9] F. Belli. Finite-state testing and analysis of graphical user interfaces. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, pages 34–43, Washington, DC, USA, 2001. IEEE Computer Society.

[10] D. Bjørner and C. B. Jones. The Vienna Development Method: The meta-language. In *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[11] A. Blandford, R. Butterworth, and P. Curzon. PUMA footprints: linking theory and craftskill in usability evaluation. In M. Hirose, editor,

Proceedings of Interact: 8th TC13 IFIP International Conference on Human-Computer Interaction, pages 577–584. IOS Press, July 2001.

[12] A. Blandford, R. Butterworth, and P. Curzon. Models of interactive systems: a case study on programmable user modelling. *International Journal of Human-Computer Studies*, 60(2):149–200, 2004.

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

[14] J. Bowen and S. Reeves. Formal models for informal GUI designs. In *1st International Workshop on Formal Methods for Interactive Systems, Macau SAR China, 31 October 2006*, volume 183, pages 57–72. Electronic Notes in Theoretical Computer Science, Elsevier, 2006.

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

[16] J. Bowen and S. Reeves. Refinement for user interface designs. In P. Curzon and A. Cerone, editors, *Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems (FMIS 2007)*, volume 208, pages 5–22, Lancaster University, UK, September 2007. Electronic Notes in Theoretical Computer Science, Elsevier.

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

Conference(HCI 2007, University of Lancaster, UK), volume 1, pages 159–166. British Computer Society, 2007.

[18] J. Bowen and S. Reeves. Formal models for user interface design artefacts. *Innovations in Systems and Software Engineering*, 4(2):125–141, 2008.

[19] C. Bramwell. *Formal development methods for interactive systems: Combining interactors and design rationale*. PhD thesis, The University of York, 1995.

[20] C. Bramwell, R. E. Fields, and M. D. Harrison. Exploring design options rationally. In P. Palanque and R. Bastide, editors, *Design, Specification and Verification of Interactive Systems '95*, pages 134–148, Wien, 1995. Springer-Verlag.

[21] P. Bumbulis, P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena. Combining formal techniques and prototyping in user interface construction and verification. In P. Palanque and R. Bastide, editors, *Design, Specification and Verification of Interactive Systems '95*, pages 174–192, Wien, 1995. Springer-Verlag.

[22] R. Butterworth and A. Blandford. Programmable user models: the story so far. Puma working paper WP8, Middlesex University, 1997.

[23] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems (The International Series on Discrete Event Dynamic Systems)*. Springer-Verlag, September 1999.

[24] L. Chubb. *Data Refinement Models for Program and User Interfaces*. PhD thesis, University of New South Wales, 1995.

[25] P. Curzon and A. Blandford. From a formal user model to design rules. In *DSV-IS '02: Proceedings of the 9th International Workshop on Interactive Systems. Design, Specification, and Verification*, pages 1–15, London, UK, 2002. Springer-Verlag.

[26] M. Cusumano, A. Maccormack, C. F. Kermerer, and W. Crandall. A global survey of software development practices. Technical Report 178, MIT Sloan School of Management, June 2003.

[27] Community Z tools project. Community Z Tools Project Sourceforge pages:
`http://czt.sourceforge.net`.

[28] F. de Rosis, S. Pizzutilo, and B. de Carolis. Formal description and evaluation of user-adapted interfaces. *International Journal of Human-Computer Studies*, 49(2):95–120, 1998.

[29] N. Delisle and D. Garlan. A formal specification of an oscilloscope. *IEEE-SOFTWARE*, 7(5):29–36, 1990.

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

[31] E. W. Dijkstra. Notes on structured programming. In *Structured Programming*. Academic Press, 1969.

[32] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[33] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall, 3rd. edition, 2004.

[34] A. Dix and C. Runciman. Abstract models of interactive systems. *People and Computers: Designing the Interface*, pages 13–22, 1985.

[35] G. J. Doherty and M. D. Harrison. A representational approach to the specification of presentations. In *Eurographics Workshop on Design Specification and Verification of Interactive Systems, DSVIS 97, Granada, Spain*, pages 273–290, 1997.

[36] D. J. Duke and M. D. Harrison. Abstract interaction objects. In *Proceedings of Eurographics '93, Computer Graphics Forum*, pages 25–36. NCC Blackwell, 1993.

[37] D. J. Duke and M. D. Harrison. Mapping user requirements to implementations. *IEE/BCS Software Engineering Journal*, 1(10):13–20, 1995.

[38] J. Eisenstein and A. Puerta. Adaptation in automated user-interface design. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 74–81. ACM Press, 2000.

[39] M. Evers. Adaptability problems of architectures for interactive software. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 284–290, London, UK, 1999. Springer-Verlag.

[40] G. Faconti and F. M. Paternò. An approach to the formal specification of the components of an interaction. In C. Vandoni and D. Duce, editors, *Proceedings of Eurographics '90. North Holland*, pages 481–494. Springer, 1990.

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

[42] M. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer, 1984.

[43] J. T. Hackos and J. C. Redish. *User and task analysis for interface design*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[44] M. D. Harrison and D. J. Duke. A review of formalisms for describing interactive behaviour. In *ICSE '94: Proceedings of the Workshop on Software Engineering and Human-Computer Interaction*, pages 49–75, London, UK, 1995. Springer-Verlag.

[45] M. D. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction*. Cambridge University Press, New York, NY, USA, 1990.

[46] M. C. Henson and S. Reeves. A logic for the schema calculus. In J. P. Bowen and A. Fett, editors, *Proc. Int. Conf. on Z: ZUM '98 LNCS 1493*, pages 172–191. Springer, 1998.

[47] M. C. Henson and S. Reeves. New foundations for Z. In J. Grundy, M. Schwenke, and T. Vickers, editors, *Proc. International Refinement Workshop and Formal Methods Pacific '98*, pages 165–179. Springer, 1998.

[48] W. E. Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, 4:11–26, 1952.

[49] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

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

[51] J. Jacky. *The Way of Z: Practical programming with formal methods*. Cambridge University Press, 1997.

[52] B. E. John, K.Prevas, D. D. Salvucci, and K. Koedinger. Predictive human performance modeling made easy. In *CHI '04: Proceedings of*

the SIGCHI conference on Human factors in computing systems, pages 455–462, New York, NY, USA, 2004. ACM.

[53] C. W. Johnson. A probabilistic logic for the development of safety-critical, interactive systems. *International Journal of Man-Machine Studies*, 39(2):333–351, 1993.

[54] C. W. Johnson. Using Z to support the design of interactive safety-critical systems. *IEE/BCS Software Engineering Journal*, 10(2):49–60, 1995.

[55] H. Kaindl and R. Jezek. From usage scenarios to user interface elements in a few steps. In C. Kolski and J. Vanderdonckt, editors, *Proceedings of 4th International Conference of Computer-Aided Design of User Interfaces, CADUI'2002*, pages 91–102. Kluwer, 2002.

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

[57] J. Landay and B. Myers. Just draw it! programming by sketching storyboards. Technical Report CMU-CS95, Carnegie Mellon University, School of Computer Science, 1995.

[58] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. UsiXML: A language supporting multi-path development of user interfaces. In *Proc. of 9th IFIP Working Conf. on Engineering*

for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems, EHCI-DSVIS'2004, pages 200–220. Kluwer Academic Press, 2004.

[59] K. Loer and M. D. Harrison. Formal interactive systems analysis and usability inspection methods: Two incompatible worlds? In *7th International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS 2000)*, volume 1946, pages 169–190. Lecture Notes in Computer Science, Springer Verlag, 2000.

[60] K. Loer and M. D. Harrison. Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In W. Emmerich and D. Wile, editors, *Proceedings 17th International Conference on Automated Software Engineering*, pages 223–226. IEEE Computer Society, September 2002.

[61] K. Loer and M. D. Harrison. An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation. *Automated Software Engineering*, 13(4):469–496, 2006.

[62] R. Mahajan and B. Shneiderman. Visual and textual consistency checking tools for graphical user interfaces. *IEEE Trans. Software Engineering*, 23(11):722–735, 1997.

[63] Microsoft Visual Studio. Microsoft technical pages for the Visual Studio Software:

http://msdn2.microsoft.com/en-us/vstudio/default.aspx.

[64] R. A. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

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

[66] D. Norman. *The Psychology of Everyday Things*. Basic Books, April 1988.

[67] A. Paiva, J. C. P. Faria, and R. F. A. M. Vidal. Specification-based testing of user interfaces. In J. A. Jorge, N. J. Nunes, and J. F. e Cunha, editors, *Interactive Systems. Design, Specification, and Verification, 10th International Workshop, DSV-IS 2003*, volume 2844, pages 139–153. Lecture Notes in Computer Science, Springer, 2003.

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

[69] P. Palanque and F. M. Paternò. *Formal Methods in Human-Computer Interaction*. Springer-Verlag New York, Inc., 1997.

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

[71] F. M. Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, UK, 1999.

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

[73] F. M. Paternò, G. Mori, and C. Santoro. Tool support for designing nomadic applications. In *Proceedings of 8th International Conference on Intelligent User Interfaces IUI'03*, pages 141–148. ACM, 2003.

[74] F. M. Paternò and C. Santoro. One model, many interfaces. In C. Kolski and J. Vanderdonckt, editors, *CADUI 2002, volume 3.*, pages 143–154. Kluwer Academic, 2002.

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

[76] S. J. Payne and T. R. G. Green. Task-action grammars: A model of the mental representation of task languages. *Human-Computer Interaction A Journal of Theoretical, Empirical, and Methodological Issues of User Science and of System Design*, 2(2):93–133, 1986.

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

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

[79] G. Reeve. The syntax and semantics of μ-charts. Technical Report 04/2004, Department of Computer Science, University of Waikato, 2004.

[80] G. Reeve. *A Refinement Theory for μCharts*. PhD thesis, The University of Waikato, 2005.

[81] S. Reeves and D. Streader. General refinement, part one: interfaces, determinism and special refinement. *Proceedings of Refine 2008, Electronic Notes in Theoretical Computer Science*, 2008.

[82] S. Reeves and D. Streader. General refinement, part two: flexible refinement. *Proceedings of Refine 2008, Electronic Notes in Theoretical Computer Science*, 2008.

[83] M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM '97: The Z Formal Specification Notation. 10th International Conference of Z Users. Proceedings*, pages 72–85, Berlin, Germany, 3–4 1997. Springer-Verlag.

[84] P. Scholz. An extended version of mini-statecharts. Technical Report TUM-I9628, Technische Univerität München, 1996.

[85] A. Shepherd. Analysis and training in information technology tasks. In D. Diaper, editor, *Task Analysis for Human-Computer Interaction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1990.

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

[87] C. Snyder. *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*. Morgan Kaufmann, 2003.

[88] I. Sommerville. *Software engineering (4th ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[89] Spec #. Microsoft technical pages for Spec #: `http://research.microsoft.com/specsharp/`.

[90] S. Stepney, D. Cooper, and J. Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[91] B. A. Sufrin. Formal methods and the design of effective user interfaces. In M. D. Harrison and A. F. Monk, editors, *People and Computers: Designing for Usability*. Cambridge University Press, 1986.

[92] Tcl Developer Xchange

http://www.tcl.tk/.

[93] H. Thimbleby. Reflections on symmetry. In *Complementizers and WH Constructions*, pages 28–33. Springer Verlag, 2002.

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

[95] H. Thimbleby and P. Ladkin. From logic to manuals again. *IEE Proceedings - Software*, 144(3):185–192, 1997.

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

[97] J. Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly, Cambridge, 2006.

[98] Top 10 mistakes in web design, available online. Jakob Nielsen's usable information technology website:

http://www.usit.com.

[99] User interface extensible markup language. Homepage for the USIXML project:

http://www.usixml.org/.

[100] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Morgan and Kaufmann, 2006.

[101] J. A. van der Poll, P. Kotzé, A. Seffah, T. Radhakrishnan, and A. Alsumait. Combining UCMs and formal methods for representing and checking the validity of scenarios as user requirements. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablment through technology*, pages 59–68, Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.

[102] K. Vredenburg, J-Y. Mao, P. W. Smith, and T. Carey. A survey of user-centered design practice. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 471–478, New York, NY, USA, 2002. ACM Press.

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

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

[105] XAML.net Information Source `http://www.xaml.net/index.html`.

[106] eXtensible Interface Markup Language. Homepage for the XIML project: `http://www.ximl.org/`.

[107] XML User Interface Language (XUL)

`http://www.mozilla.org/projects/xul`.

[108] R. M. Young, T. R. G. Green, and T. Simon. Programmable user models for predictive evaluation of interface designs. *SIGCHI Bulletin*, 20(SI):15–19.