A formal framework for the specification of interactive systems

by

Richard Butterworth

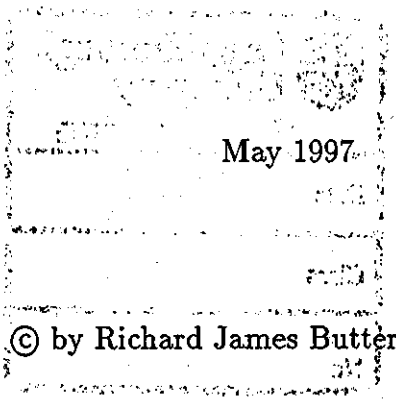
B. Sc. Combined Science (Hons) University of Leicester (1993)

A Doctoral Thesis

Submitted in partial fulfilment of the requirements

for the award of

Doctor of Philosophy of Loughborough University



A formal framework for the specification of interactive systems

by

Richard Butterworth

Submitted to the Department of Computer Studies
on 17th May, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We are primarily concerned with interactive systems whose behaviour is highly reliant on end user activity. A framework for describing and synthesising such systems is developed. This consists of a functional description of the capabilities of a system together with a means of expressing its desired 'usability'. Previous work in this area has concentrated on capturing 'usability properties' in discrete mathematical models.

We propose notations for describing systems in a 'requirements' style and a 'specification' style. The requirements style is based on a simple temporal logic and the specification style is based on Lamport's Temporal Logic of Actions (TLA) [74]. System functionality is specified as a collection of 'reactions', the temporal composition of which define the behaviour of the system.

By observing and analysing interactions it is possible to determine how 'well' a user performs a given task. We argue that a 'usable' system is one that encourages users to perform their tasks efficiently (i.e. to consistently perform their tasks well) hence a system in which users perform their tasks well in a consistent manner is likely to be a usable system.

The use of a given functionality linked with different user interfaces then gives a means by which interfaces (and other aspects) can be compared and suggests how they might be harnessed to bias system use so as to encourage the desired user behaviour. Normalising across different users and different tasks moves us away from the discrete nature of reactions and hence to comfortably describe the use of a system we employ probabilistic rather than discrete mathematics.

We illustrate that framework with worked examples and propose an agenda for further work.

Keywords : Formal specification, interactive systems, usability, human-computer interaction, reactive systems, system use.

Thesis Supervisor: Dr D. John Cooke
Title: Lecturer

Contents

Acknowledgements

First and foremost I must acknowledge my supervisor Dr. John Cooke without whose ideas, patience, forebearance, humour and willingness to answer even the most ridiculous questions again and again and again, far beyond the call of duty, this thesis would quite simply not have been written.

I would like to acknowledge the time spent on a purely voluntary basis by Dr Ann Blandford, Prof Harold Thimbleby and Dr Gordon Rugg who commented on and proof read the various sketchy, half baked drafts that I showed them. Dr Blandford, Dr David Duke and Dr Richard Young further deserve mention for having the mixture of visionary foresight and blind desperation for employing me on the strength of a ten minute panic stricken presentation and an application form recently nominated for a creative writing prize. Jason Good deserves mention for sympathy, tolerance and tea offered in the Dark Days of Write Up. Dr Blandford gets another mention (making a record breaking three in all) for tolerantly allowing me the time off work while I dithered about getting the write up finished.

The You-Are-All-Wonderful-In-No-Particular-Order list is as follows: Lynne Evans, Tom Ahnk-Jans-Uhmn, Linda Hodges, Vanessa Manship, The Hairy Stupids, Dr Jon 'Can you fix my PC for me' Knight, Martin Hamilton, Neil Bramhall, Charlie Foulkes, McAndrew McKane, McKaren 'McParochial' McKenzie McMills, Karl Perkins, McKaren's McKettle, Diana Climpson, Jenny Ardley, Louisa Allen, The 1995/1996 Community Action committee, especially Helen Bradshaw and Phil Tipper, The Loughborough Student Union advice centre, Mum and Dad and the rest of the expansionary Butterworth family. The situated memory music for three years in Loughborough was Brave by Marillion. 'Tell me I'm mad. I have been here for so long.' Last, but by no means tallest, Stuart 'Three Degrees' Sutherland (BSc, PhD, PGCE) gets in my acknowledgments for no better reason than I got in his.

This thesis was typeset using LaTeX2e using Jon Knight's lutthesis style. I learnt to use LaTeX without a graphical previewer. I apologise to all the trees that died as a result.

Dedication

To Joseph Brown (1903–1996)
who, I hope, would have been proud.

Chapter 1

Introduction

This thesis reports a framework that is intended to help in the synthesis of interactive systems. An interactive system is a system the behaviour of which is heavily reliant on end user activity. The framework is mathematical in nature and should allow design decisions to be made based on abstract models of systems and then these models developed into actual implementations.

Typically the design of interactive systems relies heavily on the craft skill of human factors experts. It has been argued that such an approach to design is unsatisfactory [79] on the grounds that the knowledge of human factors experts does not transfer or generalise. On the the other hand it has been argued [30] that the design of computerised systems that work well with humans is a human endeavour, unsuited to mathematical rigour. Attempts to bring a design process for interactive systems (where typically an interactive system is designed and implemented then tested for usability against a user population) into a more theoretical domain may be misguided. We review such arguments in more detail in chapter 3 and come to the conclusion that there is a sensible middle ground between these two arguments — developing interactive systems using informal and heuristic techniques as well as developing these heuristics into more explicit formalisations so that they can be reused in a generalised way.

The primary motivation of this work is to describe a formal framework for synthesizing interactive systems which allows for theoretical information to be fed into the design process. However there is little (in any) such information readily available in the relevant literature. We argue that repeated use of the framework would play a role in evolving heuristics and craft skill into explicit theory which can be then used for synthesizing interactive systems.

We intend to suggest a design process based on the framework. We hope that the clarity introduced by capturing an interactive system formally may inform other attempts to capture design processes. We argue that our work is novel in the way it formally captures what it means for a system to be usable and what makes a good user interface. For a design process based on this framework to be a success it needs to integrate and draw upon previous work, not sit in isolation or completely supercede previous work. Our framework intends to take the good, rigorous practice of a formal approach and attempt to apply these practices to usability considerations and the synthesis of user interfaces.

1.1 On unlikely marriages: the curious case of formal methods and HCI

The framework attempts to integrate human issues — the field of human-computer interaction — with mathematical modelling techniques — the field of formal methods. We start by outlining these two fields.

1.1.1 What are formal methods?

'Formal methods' is the commonly (and wrongly[1]) used term to describe a highly mathematical way of describing and producing computer systems and in particular computer software systems. The benefit of a formal approach is that it allows for a high degree of certainty that a software system produced in a formal way is correct. However, formal methods are held to be confusing (which, as we shall argue later, should not be the case) and expensive (which is true in the short term).

Formal approaches have been studied and described very thoroughly in academia and there is now the beginnings of a trickle-down into industry. Formal methods' impact on industry is limited to areas that produce safety-critical software; correctness being essential in such areas. It is not difficult to catalogue a whole host of examples where software failure has had expensive and even life-threatening results (a recent and dramatic incident being the failure of the Ariane

[1] 'Formal methods' being one part of a formal approach to computer software systems. Henceforth we shall endeavour to use the term 'formal approaches' when referring to the whole field.

mission) and where a formal approach would have most likely prevented such failures.

It is held ('its generally accepted that user interface conformance resides in the domain of informal reasoning' [20]) that only certain parts of the software system will benefit from being produced using a formal approach. This argument is debatable; a chain only being as strong as its weakest link. The nature of formal approaches means that there is a steep learning curve for analysts and designers wishing to use them. We suggest that it is this learning curve that is preventing a more thorough and wholesale acceptance of formal approaches in industry.

One of the areas of system design for which formal approaches are held to have little (or even no) relevance is the design of user interfaces and other such 'human' areas. We will contest this view.

1.1.2 What is HCI?

Human-computer interaction (HCI) is the study of the user side of interactive systems; it is a wide field, bringing in ideas from psychology and sociology as well as computer studies. In fact there seems to be no limit to how far HCI workers are willing to cast their nets for insight and inspiration into their field; see [78] for how even drama and theatre studies has been plundered for ideas. This diversity in HCI means that it is one of the most wide ranging and challenging fields in computer science, it also means that it is one of the most frustrating — despite many advances and insights provided by HCI work, the field is ragged and loose. There is unfortunately little effort to bring its findings into a more rigorous and inspectable whole. It is therefore seen in many quarters as a woolly and imprecise affair and this view hinders its inclusion into other areas of computer science. This is a very lamentable state of affairs; in many cases the user interfaces of widely available and heavily used computers systems are confusing and difficult. The computer revolution in our society continues in spite of, not because of the interfaces that are built into computer systems.

So what potential benefits are there in bringing the two fields of HCI and formal methods together? In the preface to 'Formal methods for interactive systems' [41] Alan Dix expresses the opinion that formal approaches and HCI are at first sight an 'unlikely marriage'. We are not the first to advocate a formal approach to the analysis and design of interactive systems, indeed there is now a considerable body of work in the area and a regular conference circuit (a sure

sign of increasing maturity). So, before continuing our work we must ask ourselves whether the marriage is working out or it is time to call in the lawyers and settle an amicable divorce.

1.1.3 Justifying a formal approach to HCI

We justify a formal approach to HCI on simple and clear grounds. Formal approaches reason about *what* systems do, not *how* they do it, they abstract away from all the clutter and detail of systems and concentrate on the germane, critical and interesting features of the system. This is also a user's view of a system. A user will have little knowledge or interest in the intricacies of how a computer systems works — they only want to know what a system does and how that fits in with the task they are wishing to use it for. Formal, abstract approaches support this view of interactive systems.

Furthermore, Thimbleby [114] vigorously justifies the marriage of formal approaches and HCI from both the designer's and user's viewpoint.

Good formal approaches facilitate clear and creative thinking, in particular they:

- communicate crisp ideas,
- expose ideas for criticism and improvement,
- delineate what is relevant to reasoning,
- expose irrelevant, inconsistent or fallacious arguments,
- permit reasoning by correspondence and
- make general arguments.

Thimbleby asserts that these advantages can not only be used in HCI, but are 'desperately needed'. Applying formal approaches to HCI moves the critical point of design to an earlier stage of the system life-cycle. Thimbleby describes this as HCI's 'try it and see' dictum contrasted to formal approaches' 'think and understand' maxim.

Users work to rules and guide-lines when using a system; a mouse pointer should move in normal Euclidean space. Formal approaches are adept at defining such rules. Hence the application of formal approaches will ensure that the user's view of a system is regular and consistent (although consistency is becoming a much abused term in HCI research.)

1.1.4 Why some previous approaches have been less than successful

Having expounded the potential benefits of a formal approach to HCI, we should now look at the actual experience in the past decade and try to catalogue some of the pit-falls.

The intended audience

In many cases formal approaches to HCI are suggested but their intended uses are not made sufficiently clear to the intended audience, indeed the intended audience is often not identified either. Hence confusion arises — a formal approach to HCI should not be about presenting confusing formalisms to HCI practitioners or obtuse psychological theories to software engineers. We should always be aware that the average software engineer and HCI practitioner will know and care little for each others' fields of work and, what is more important, they should not need to.

System or software design?

The distinction between system and software design is rarely made clear. System design concerns the whole system; bits of hardware, potential user populations *etc.* whereas software design is a proper sub-set of system design — a software design does not constitute a whole system design. Whether usability requirements should be part of the system or software design is not clear.

Is formality *really* necessary in an approach?

More subtly, we suspect that several researchers are not clear about exactly why they are applying formalisms to HCI. It is clear that certain aspects of HCI can be formalised, but doing something simply because it can be done is no justification. A formalism should not sit in isolation from the system design process — it should be a useful part of the process unless we consider a formal approach to HCI to be a wholly academic exercise. Wholly academic exercises are by no means a bad thing, but it should be made clear if a formalism has a practical purpose (and, of course, what that purpose is) or whether it is purely academic.

Notations and languages

Lastly, in our introductory catalogue of confusion, we can see that the plethora of formal notations available also causes problems. It is well argued that the general, all-purpose formalism languages such as Z and VDM are not especially well suited to formalising interactive systems. The enterprising formalist therefore feels justified in taking one of several paths — adding extra bits of formalism onto existing languages, twisting existing languages to directions they were not really meant for or simply making up a whole new formalism to suit their purposes. In the ideal world none of these approaches is really satisfactory.

1.1.5 Summarising the arguments — good content, bad presentation

In chapter 3 we will present arguments that a formal approach to HCI is *fundamentally* flawed; that HCI is simply too unpredictable and involved for the 'clinicism' of a formal approach to be of any use. We will argue that such arguments are invalid (otherwise this would be a lamentably short and/or pointless thesis).

We believe the problem with a formal approach to HCI has been its application; there is nothing 'wrong' with many of the formalisms and models proffered in the literature, their intentions and uses have not been made clear. In general it is the presentation rather than content that has been at fault.

1.2 Our intentions in this work

Maybe it is as well to point out here that this work stems from formalists venturing into HCI rather than the other way around. Our work is therefore going to be 'biased' to formal approaches. But we contend we will demonstrate how we can offer significant bridges across the divide to HCI workers.

Of course our aim is to try to capture the benefits formal methods can bring to HCI without slipping into the pitfalls we have catalogued. However, such a task is well beyond a single PhD thesis and so we must limit our aims, and crucially we must make those limits explicit.

This section is a very broad look at our intentions — we discuss these ideas in more detail and depth in chapter 2.

1.2.1 Developing and applying a theory of HCI

In chapter 3 we will discuss theories of HCI. For our purposes a theory is an encapsulation of ideas about HCI that can be used by system developers, during (rather than after) the development process, to improve the usability of interactive systems. Unfortunately such theory is thin on the ground and many HCI decisions are made very late in the development process in an *ad hoc* manner.

Our main motivation is the development of a framework which can guide the synthesis of interactive systems *i.e.* a framework which takes existing HCI theory and feeds its conclusions into the design process.

Much of the previous formal work in HCI has described usability issues in a way that is fairly disjoint from a development process. Our intention is to propose a development process and place formally stated usability issues within this process; suggesting how the insights gained are incorporated into the process and feed through into the eventually implemented system.

However the assumption that there exists a body of theory that we can simply 'plug into' our framework is unrealistic. Our framework should aim to make the reasoning behind design decisions transparent. However in many cases there will not be enough theory available to guide design decisions in any reliable manner.

The use of a framework that helps to lay bare the context for a decision should help in the development of a theory. If a designer finds himself repeatedly making the same decisions in the same context with the same result then he would be in a strong position to argue that some HCI 'micro-theory' has been exposed. If however the designer finds that the same decision causes different results in apparently the same context then he would be able to argue that the description of the context in not expressive enough for that decision.

Our intended development process should allow for the functionality of an interactive system to be specified and developed towards an implementation. We also intend that non-functional usabilty aspects be developed in a process that runs parallel to the development of the functionality. The usability issues are modelled by considering how a system is expected to be used, how the designers want the system to be used and what user interface features move the former closer to the latter.

1.2.2 Characterising the user-interface

What is a user-interface? More importantly what is the *effect* of an interface? We propose a model of user-interfaces that concentrates on the effect they have on the behaviour of a system. We can then discuss *what* we want of a user-interface, abstracting away from the mechanics of the interface. The idea being that the system designer can specify the effect on the system behaviour that he requires the user-interface to have and pass this specification to a human factors specialist who knows (somehow) what actual interface features will fulfill this specification.

Unfortunately we know of little work that would guide the transition from what a user interface does as captured by our framework to how it is done. Here we see one example of how our framework should aid in the firming up of HCI theory. At the moment the step from abstract mathematical models of what an interface does to actual interface features would rely (at best) on the craft-skill of human factors specialists in interpreting those models. However if a human factors expert repeatedly found himself successfully implementing the same interface features to achieve the same result then we can argue that this exposes some HCI theory.

1.2.3 Horses for courses and notations for practitioners

There is a suggestion in the literature and formal HCI community that a utopian state of affairs would be achieved with the introduction of a 'universal' formal notation that is suitable to all practitioners in the field. We are suspicious of this philosophy; we hold notations lightly in that a notation is simply a way of expressing a model of a system. It is those models and what we do with them that is crucial, not what notations they are expressed in. Although we develop and propose notations in this work, we do not claim them to be 'better' than other notations. Our notations are simply tools for talking about models; we can envisage a hypothetical thesis being written in parallel to this one using a different set of notations and producing the same results.

We therefore intend to describe notations for requirements engineers, system specifiers and propose how they relate to more traditional software engineers notations such as VDM. We also look at graphical notations for HCI workers for use in the more user-centred areas of system design.

1.2.4 Mathematical notations

Much of the work in this thesis concerns the proposing and description of mathematical entities and describing relationships between them. To do this we use set theoretic mathematics and predicate logic. We assume that the reader is familiar with basic set theoretic and logical notations. A catalogue and formal description of the notation we use is included in the mathematical appendix (section B).

We describe relationships between mathematical entities with functions which are typically described by a signature, a definition section and a description of the function in English as follows...

$$\begin{array}{l} f : X \to Y \\ f(x) \mathrel{\hat{=}} F(x) \end{array} \tag{1.1}$$

In words; 'the function f takes values of type X to values of type Y. For all values x of type X the result of $f(x)$ is defined to be $F(x)$ (assuming that F is defined elsewhere).'

A special case of a function is a 'predicate' which takes values from a domain to members of the Boolean set $\{\mathsf{true}, \mathsf{false}\}$.

1.2.5 Terminology

A potential cause of much confusion in a 'cross-over' area such as a formal approach to HCI is the loose use of terminology. There have been several subtly different models presented in the literature described using many variations of terminology. When we introduce a term that may have an ambiguous meaning (or may have been used in previous work to mean something else) we enclose it in quotes and explain it as unambiguously as possible afterwards. We also include a glossary of terminology used in the appendix (section A).

1.3 Thesis chapter plan

1. Introduction

The fields of HCI and formal methods are briefly introduced and the potential benefits and pitfalls of a formal approach to HCI are discussed. There is an outline of our intentions for this

work and a brief plan for the thesis.

2. The objectives and aims of this work

This work aims to be part of a convergence in formal HCI; borrowing good ideas from previous work and introducing our own novel approaches.

We aim to produce a formal framework for the construction of more usable interactive systems. As well as describing what we intend to do, we carefully delineate what we *do not* intend to do.

3. Constructing usable systems and the 'theory of HCI debate'

To make decisions about usability and other related HCI phenomena early in the system design life-cycle means we assume that HCI is backed by a body of abstract theory. HCI obviously is not and the advisability of creating such a body of theory has been hotly debated. We look at this debate and conclude that although there is little HCI theory, such a body of knowledge would be advantageous to the field. We look at some proposed frameworks for HCI; a framework essentially being the structure of a theory into which we would 'slot' HCI knowledge when and if it is 'discovered'.

4. A review of previous work in the field

First of all we look at the various formal notations and techniques that have come to prominence in computer science. We look at several of the specification languages in general use and conclude that they lack the expressive power we require for our approach. Some of the reasons for this are discussed.

This is by no means the first attempt to bring formal methodologies to bear in HCI. We look at the previous work concentrating on the 'York approach' to interactive system design as we believe this will have the most relevance to our approach. We conclude by discussing some of the more notable 'holes' in the literature.

5. Introducing a Reactive System Specification Language (RSSL)

We describe a typical system design process and describe a language for describing systems within this design process. We start with a very simple system and add layers of complexity incrementally, introducing the features of our language one by one. Our technique covers requirements engineering, system specification and design, software system design and software engineering. We also look at some suggestions for implementations and program derivation.

6. A formal semantics for RSSL

Having introduced our specification language 'by stealth' in the previous chapter we now give a full semantics for it. We divide the language into two; the temporal logic for expressing requirements and a specification language based on the Temporal Logic of Actions (TLA) [74]. We define a model of real-time system behaviour and give the syntax and semantics for both languages in terms of this model. We also look at a syntactic sugar for the specification language that allows us to specify reactive systems more comfortably.

7. Describing the use of an interactive system with an Interactive System Specification Language (ISSL)

So far we have described systems using discrete mathematics; either behaviour is legal or it is not. We argue that this is a rather arid way of describing systems containing highly non-determistic users. We propose some ideas for modelling the system using probabilistic mathematics so as to better capture the user behaviour.

This proposal opens up a whole host of ideas about interactive systems which we look at and discuss. Most importantly we can characterise a user-interface in terms of what it does to an interactive system without worrying about how it does it.

8. A formal semantics for ISSL

The specification language RSSL does not have the tools to comfortably describe probabilistic behaviour. We therefore look at how we can bring probabilistic models into our language so as adequately describe interface behaviour.

We also go into some more formal depth with some of the ideas discussed in the previous chapter.

9. Some examples of our technique in use

We work though several examples using our technique, hopefully displaying its ability to cope with examples previously presented in the literature and to bring more expressive power to bear on these examples.

10. Conclusions and an agenda for future work

Finally we summarise our approach to formally synthesizing interactive systems and propose ways of expanding on our work.

Chapter 2

The objectives and aims of this work

Of course the aim of any thesis is to push the boundaries of the relevant discipline. Whilst we contend that this work is novel in its approach of bringing HCI considerations into a formal system synthesis process, that is not to say that we do not intend to borrow, develop and integrate into our approach many of the ideas and techniques that have been previously presented in the literature.

There follows a list of areas of concern that we intend to cover in detail in this work.

2.1 Interactive systems as reactive systems

A reactive system is generally held to be a computerised system that reacts continuously to externally generated events. We widen this definition rather, viewing a reactive system as the conjunction of the computerised 'kernel' and its 'environment' not just the central machinery itself.

Several writers [113, 83] have recently suggested that interactive systems are a sub-class of reactive systems. We conform to this view — the kernel is the computerised machinery and the environment the user population. The users employ the computer as a tool in the accomplishment of some tasks. There is therefore an imbalance between the kernel and environment; the users 'control' the computer by issuing requests that the computer must respond to, but not *vice versa*.

Typical in reactive and interactive systems are concurrent streams of behaviour that occur

through multiple modalities. For example this thesis is being written on an interactive system, which provides the writer with four different windows; one to edit the source text, one to compile the text into type-set text, one to view the type-set text and one to deal with various subsidiary maintenance matters such as editing the bibliographic database or viewing previous drafts of the work and rough notes. Each of these windows represents streams of interaction with the underlying functionality, these streams interleave and are interdependent. The writer manipulates these windows using a keyboard and mouse only, but there is little (other than cost) preventing him from using other devices in other modalities such as voice recognition. The computer responds to the commands issued by the writer in both the visual and auditory modalities. This is just one example of a fairly simple task being performed interactively on a computer system. It is not difficult to find examples of systems that use much more involved interactions than this.

It is easy to become intimidated by this high level of complexity and many workers in the field have proposed increasingly complex notations to be able to capture this interactivity.

We assert, however, that we can cut through the clutter with abstraction. The important point for us is initially not to worry about about the complexity of different interface presentation issues and modalities but to consider how we would prefer the system (the 'system' being both the computer and user) to behave. The complexity should come into the system synthesis process only when needed.

For example if we were building from scratch a system to help a writer to type-set his thesis then our first considerations would be the writer's task itself and what functionality is needed to achieve that task. Only then do we begin to think about how that functionality is presented to the writer in such a way that it is helpful to him and helps him get his task done efficiently. Decisions about having four windows are very low level; it is to do with 'how' the system is implemented, not 'what' the system is doing. As formalists we are initially (much) more interested in the what rather than the how.

We will introduce a notation that allows us to describe the requirements of a reactive system in a very abstract way, then we shall introduce a model of reactive systems where the kernel offers a collection of 'reactions' to the environment. A reaction is a pairing of some environment functionality with some kernel functionality, such that there is causality from the environment

to the kernel. The environment has a non-deterministic choice between these reactions so the system behaviour is (at its simplest) all possible sequences of reactions.

2.2 Including usability as part of the system synthesis process

It is well argued that important design decisions should be made early on in the system design process. If a mistake is made and identified early in the design process then rectifying is much cheaper than if mistakes are made later on. We assert that usability issues are crucial to interactive systems and therefore decisions effecting usability should be made as early in the interactive system design process as possible.

Currently most HCI work takes place towards the end of the design process. Our approach suggests how we can push usability issues higher up the process.

However, before we can start to think about such matters we need to be clear about the design process itself. We therefore begin our work by looking at and illustrating a design process for reactive systems and considering what extra apparatus and ideas are needed when we specialise the process to interactive systems. Of course we are not talking about 'specialisation' in its strict sense. If interactive systems were a strict specialisation of reactive systems then we would need no more apparatus to describe interactive systems than we would to describe reactive systems. The real world is not that simple however.

2.2.1 Synthesizing reactive systems

Recall that we defined a reactive system as being the conjunction of a 'kernel' of automated functionality and its environment. We define a system as being the conjunction of the two so as to make explicit the assumptions we are making about the environment.

We deal by preference with 'closed' systems which are systems with no external influences; all their behaviour is determined by their constituent sub-systems. A common way of discussing open systems [69] is to describe 'assumption and guarantee' pairs, such that an open system guarantees a property if its environment satisfies the assumption. Working with closed systems forces us to preemptively describe all the assumptions about a system and it therefore becomes clear where assumptions are coming from and what their rationale is. In other words it gives us

a context for our description of the automated kernel to sit in.

We have also described the imbalance between the environment and the kernel in that the kernel is under some degree of obligation to respond to the environment, but there is no complementary obligation on the environment. We can draw another distinction between the two; namely that typically the environment already exists and it is the kernel that we wish to build. This is an important point to recognise in our work; we *describe* and discuss whole closed systems, but we *specify* and build only the kernel.

An often repeated theme in our work is the formula...

$$\text{requirements} \rhd \text{assumptions} \wedge \text{specification}$$

...(where $\rhd$ denotes refinement). The formula expresses that we define requirements for a system, make assumptions about the behaviour of the parts of the system which already exist and we cannot build and from these two we produce a specification of the parts of the system that need to be built. The idea being that once the proposed parts of the system that we have specified are put in conjunction with the existing parts of the system we have made assumptions about, then we will have a system that fulfills the requirements.

A direct result of this formula is that we are interested in specifying and building computer machinery.

> This work is about the specification of computer systems that certain user populations find easier to use, not about training users to find certain computer systems easier to use.

Leading on from this point, we are *not* in the business of formal (or otherwise) user modelling. We consider user modelling to be an endeavour that allows the production of rigorous and inspectable models of users that would feed into the assumptions side of our equation. We will make it clear where the results of a user modelling analysis feed into our formal framework, but we will pay little attention to actual user modelling.

2.2.2 Synthesizing software sub-systems

In saying that we are looking at building the kernels of reactive systems we are still being rather too general. What we are in fact interested in is developing sets of instructions to drive the kernel sub-systems (generally known as 'software').

2.3 Notations

In a working group on formal methods in HCI and software engineering [12] it was suggested that there were two 'deep problems' that need to be overcome before formal methods would be accepted in HCI practice.

- Firstly there needs to be the ability to transform between all the notations that abound in the development process. Ideas and insights generated in certain areas of the development process need to feed through into other areas. This could be achieved by numerous transformations between notations or a single 'universal' notation with many views.
- Secondly there needs to be models of humans that allow analysis and the prediction of behaviour. Such models need not be completely general; they only need to cover human behaviour that is involved with interaction with computers.

The second deep problem is the area of user modelling, and we have already explained that this in not our area of concern. We question the 'depth' of the first deep problem.

To be fully useful a formal notation should have rigorous semantics defined for it. Given this fact transformations between notations should be fully tractable, indeed the definition of formal relationships between objects (such as notations) is what formal methods is all about. To suggest that transformations between formal notations is intractable (or even prohibitively difficult) shows a possible miscomprehension of the fundamentals of formal methods.

The idea of a 'universal' notation is beguiling but rather utopian; a notation only becomes universal when everyone agrees to use it. The formal community is now too diverse (and set in its ways) for any one notation, no matter how wonderful, to come to dominance[1]. There

[1]Such a dominance could be argued to be very detrimental. Consider the development of programming languages, which shows many similarities to the development of formal notations. Factors other than wonder-

would still need to be transformations from this universal language to its constituent views, the creation of which would be no easier than the creation of transformations between disparate formal notations.

So the aim of this work is *not* to provide (yet) another collection of notations that we contend is 'better' than what has gone before, but to show and discuss what is done with those notations. We concentrate on the models that these notations express, showing how we develop these models from very abstract statements of system requirements to working systems.

We believe notations we present to be suitable for the task to which we put them; but we would make no further claims about them. We do not claim them to be the 'best' notations yet presented in the literature (yet we would hope that they are far from the worst); they are simply tools to describe the system synthesis process. It is entirely feasible that we might have written this thesis using an entirely different collection of notations and still shown the same results.

2.4 Viewing usability 'from above'

'Usability' is a debatable term to say the least. Informally we would say that it is a measure of how usable a given system is (or will be once constructed). If one were to place a group of respected and influential HCI workers in a room and ask them to come up with a definition of usability it is unlikely that a definition acceptable to all would be achieved. (The outcome of such an exercise would most likely be a vigorous and damning critique of the question 'what is usability?' on the grounds that it is too general and has little relevance to the process of actually making usable computer systems.)[2] Yet human factors workers and interface designers make decisions about what is and what is not usable all the time (or at least they make the easier decision of what is more or less usable).

Much of the York approach (see section 4.3) to interactive systems is about the formal definition of properties that are held to have relevance to usability. The rationale being that

fulness have decided which programming languages are dominant; currently C++ and COBOL dominate the programming languages used and one would be pushed to argue the case for wonderfulness of either candidate.

[2]We back up our hypothetical outcome to a hypothetical seminar by referring to the seminar reported in [58] in which researchers were asked 'what is consistency?' (consistency generally being held to an interface feature that greatly aids learnability) — a much smaller scale question than 'what is usability?'. The researchers were unable to give a definition other than 'Its like pornography — I can't define it, but I know it when I see it'.

usability results from the composition of several properties. If a system fulfills all the properties then the system will be usable. No adherent to the York approach would claim that they have completed the formalism of all these sub-properties, for that matter there is little clear agreement about what properties should be formalised. The properties that have been formalised tend to be held as being a bare minimum of usability — a system that upholds the properties does not necessarily ensure its usability — a system that does *not* fulfill the properties is guaranteed to be *not* usable. We view the York approach as a bottom up approach to usability; usability is the sum of several sub-properties.

We look at usability 'from above'; we describe the behaviour of a system that is usable, *i.e.* the users use the computer efficiently and productively, making an acceptable number of errors. Then we look at what we need to do with a system to make this 'optimal' behaviour more likely.

What we intend to provide is a framework in which one can ask 'in this context, what does it mean to be 'usable', what are we trying to achieve by making a system 'usable' and, once we have sorted that out, how do we construct a system that is going to fulfill that notion of usability?' We contend that it is the use of such a framework that will push HCI into the realms of theories of usability, rather than heuristics and guidelines for interface design. If whilst using a framework one were to find oneself repeatedly asking the same questions in different contexts and getting the same answers, then we would feel justified in believing we have discovered something more general than the simple accruing of data.

2.5 Towards statistical and probabilistic models

Some of the suspicions of HCI workers towards formal methods stem from the fact that formal methods work tends to be expressed in discrete mathematics whereas the mathematical language for modelling human behaviour is that of probability and statistics.

A discrete formal model tends to describe behaviour in a Boolean, black and white way and this is rather arid for expressing human behaviour. Let us return to the pertinent example of a writer composing his PhD thesis with an interactive computer system. A discrete model of this system will describe all the legal behaviour of the system. Part of that legal behaviour would be to select a window, move to the root directory and delete all the files on the hard-disk. This

is obviously extreme behaviour and does not (presumably) constitute the sort of efficient, error-free behaviour we suggested as being representative of usable systems in the previous section. A discrete model does not describe the extremity of this behaviour.

Recall from section 2.1 that we model a reactive system as the environment making a non-deterministic choice between reactions. The introduction of probability into this model allows us to describe this choice in more detail. The users (providing the environment) will be more likely to choose some reactions than others. One of the main thrusts of this work is to include statistical and probabilistic apparatus in our approach; thus providing a fairly solid bridge from the world of formal methods into that of experimental psychology. Deleting all the files on the hard-drive will be modelled in a probabilistic system as being a reaction that is legal but extremely unlikely.

However the factor that makes this behaviour so unlikely is that the user knows what he is doing and would be very loath to do such a thing. If we cannot guarantee that the user has a level of competence then we would want to construct a user interface that makes undesirable behaviour unlikely.

In introducing probabilities into the system descriptions we can develop ideas about what user interfaces do; they 'skew' the probabilistic behaviour of a system, making some behaviour more likely and some less likely. As we saw in our example the probability of a given behaviour is dependent on both the user interface and the users themselves. If we are fairly happy that the user will not accidentally perform a hard-disk delete then it is not essential that we construct an interface that will make it less likely that he will. Here we see the interplay of the 'requirements $\rhd$ assumptions $\wedge$ specification' slogan in the context of user interface. We require that it is extremely unlikely that a user deletes the hard drive, we assume that a given user is unlikely to delete the hard-drive, so the specification of the user interface does not need to make deleting the hard-drive unlikely. On the other hand if we require that it is extremely unlikely that a user deletes the hard drive and we assume that a given user is likely to accidentally delete the hard-drive then we would specify an interface that makes it difficult, and therefore unlikely that the user deletes the hard-drive.

2.6 A comparative framework

Chapter 4 shows that this is by no means the first incursion of formal approaches into HCI. To evaluate the worth of our approach against that of previous work we intend to work through several common examples of interactive systems formally described and show if our approach can cope with these examples.

A large hole in the field is currently the lack of a collection of worked case studies, although we understand that there is currently work underway to overcome this [93].

In doing so we not only hope to evaluate our own approach but to also allow judgments to be made about the worth of other approaches in the field. In this way we can begin to pick out good points, bad points and commonalities in the various approaches and hopefully move at least a little way towards the sort of scientifically valid theories we will discuss in chapter 3.

2.7 Summary

To summarise let us quickly run through what we intend to do and, just as pertinent, what we do *not* intend to do.

2.7.1 What we intend to do

- Lay down a formal framework for synthesizing reactive systems,
- concentrate on formally specifying the automated kernel of the system and developing software from these specifications,
- discuss how this framework is specialised to interactive systems,
- include probabilistic modelling tools in the approach to better capture user centred behaviour,
- capture 'what' a user interface does to the use of a system in probablistic terms, without worrying about 'how' it does it, and therefore
- propose a process for the synthesis of more usable interactive systems showing where information from the various fields of HCI come into the process.

2.7.2 What we do not intend to do

- formally model users,
- deal with 'whole system design',
- catalogue usability properties,
- provide techniques that cover the wilder shores of HCI systems, or
- attach an excessive amount of importance to the notations we employ.

Chapter 3

Constructing usable systems and the 'theory of HCI' debate

In this chapter we look at HCI, and in particular theoretical approaches to HCI in a very broad manner. Many HCI workers express suspicion with very theoretical approaches to HCI on the grounds that HCI looks at human behaviour and is therefore simply too involved, varied and context dependent to be slotted into neat scientific methodologies. We need to counter these arguments in order to be convinced that there is role in HCI for formal approaches.

3.1 What exactly are 'theories'?

The Oxford English Dictionary [106] defines a theory as being (amongst other things) a 'hypothesis ... propounded and accepted as accounting for the known facts.' If we then try to apply this definition to HCI we immediately hit several problems. Not least what we think of as 'known facts' in HCI. Facts are hard to come by in HCI literature and hypotheses accounting for them are even rarer — most of HCI work is expressed in guide-lines and heuristics which are applied *as* systems are designed and built. Ideally, however, we would like a set of guide-lines and such-like which we can apply to models of systems *before* they are built.

Predictability is then our main motivation for requiring a theory of HCI. Predictability implies there is some body of knowledge (or a theory) which lies behind HCI. If we can successfully argue that such a body exists (or can exist) then there is a genuine, useful role for formal tech-

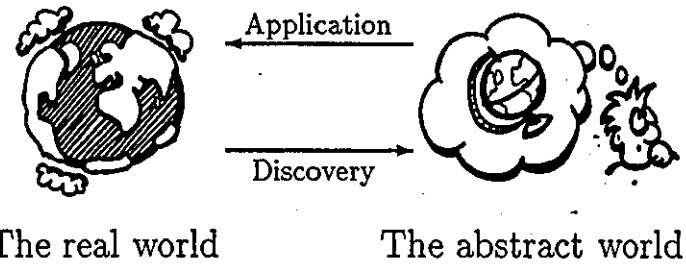


Figure 3-1: The applied science cycle

niques in building interactive systems. If we cannot, however, then formal approaches will only be useful as academic exercises.

Figure 3-1 illustrates the applied science cycle proposed by Long [80] and Barnard [9] which shows how information flows between the real world and the representational world of the 'science base'. The science base describes and explains features of the real world. A good science base will therefore also predict features in the real world. To take an example we can observe the real world behaviour of a falling weight and then in an abstract way we may make predictions about how that weight will fall in situations other than those in which we have observed it. Newton accurately predicted how a weight will fall in a vacuum without ever dropping a weight in a vacuum himself. He was able to make such predictions because he had made good observations of falling objects and had produced a sound science base from these observations.

By definition 'science' is a very wide field and we must be clear about just what area of science we are interested in and to what uses we propose to put science. We can classify science into three general categories.

Social science, for example sociology and psychology, are termed (rather detrimentally) 'soft' science by Newell and Card [87]. They are characterised by the qualitative nature of their results and their lack of formal models.

Natural science, such a physics and chemistry, that observe and model natural phenomena. The use of these models is purely explanatory and passive in the real world.

Design science, such as engineering and much of computer science, builds models from observations of the real world which are then used to construct objects in the real world, the

properties of which have been predicted in the model.

HCI manages to straddle all three categories and this may account for much of the confusion and disagreement about the role of theoretical science in HCI.

3.2 A theory of HCI?

No-one is going to argue that there exists a sound science base to HCI, ('the theories that do exist are vague and weak' [109]) what however *is* contested is whether such a base can exist and whether it would be of any use if it could. Let us look at the two main arguments for and against and then discuss each of their merits or otherwise.

3.2.1 The case for a theory of HCI

Many writers have suggested frameworks into which HCI knowledge can be slotted. For example Storrs' conceptualisation of multi-party interaction [110], Norman's cognitive engineering [89] or Long's cognitive ergonomics [80]. The utility of these frameworks can be measured by whether the sum of the knowledge in the framework is greater than the sum of the same knowledge not in the framework.

Storrs [109] justifies the need for a theory on the following grounds;

Theories give coherence to data Theories are organisations of data and as such they can show how particular data values relate to others. From these relations we can infer underlying causalities. Several theories can co-exist over common domains. It is the rejection, acceptance, strengthening or combination of these 'local' theories that mark the maturation of a field. The ultimate sign of maturity in a field is the existence of a single unified theory (as suggested by [86]).

Theories predict results. Engineering disciplines require that data results can be predicted by theories. As stated above it is this predictive quality of a theory that we are most interested in as it makes the theory useful.

Theories explain observations. Exactly what an explanation is is rather a vague notion, but if one has gained some understanding of some data then in some way that data has been

explained. In this way theories are explanations of data.

Theories develop. A theory is not simply a mass of data, it is a tool to interpret data. As our understanding of a subject increases then the theories we use to understand the subject will enhance as well. Even a wholly inadequate theory is of use to the field as it provides a reference point for research that aims to knock that theory down.

Furthermore Long [79] states that for HCI to advance as a discipline it *must* develop engineering and scientific principles to support its practices, otherwise HCI will always be seen as a 'craft' rather than a rigorous area of endeavour. Craft practices are by far the most common in the field of HCI. Dowell and Long [45] offer several criticisms of current human factors (and by implication therefore HCI) practice. Namely...

- there is little integration of HCI into system development,
- the effectiveness of HCI practices are unreliable and therefore suspect,
- HCI practices are inefficient — there is no framework into which acquired HCI knowledge may be accumulated (apart from the intuition and experience of practitioners) so experience may be lost and have to be re-acquired, and
- there is little sign of intentional progress to alleviate the above deficiencies.

Hence we have discussed both the benefits of theoretical HCI and the drawbacks of HCI practice without reference to an underlying theory.

3.2.2 The case against a theory of HCI

There is little justification in the literature for the 'craft' approach to HCI, however the simple fact that most HCI researchers do consider their field to be a craft must mean that they believe such an approach to be justified. There may also be feeling that most HCI workers want to roll their sleeves up and get on with it rather than debating probably intractable philosophical niceties. What justification there is is mostly in the form of criticising theoretical approaches (*e.g.* [77].)

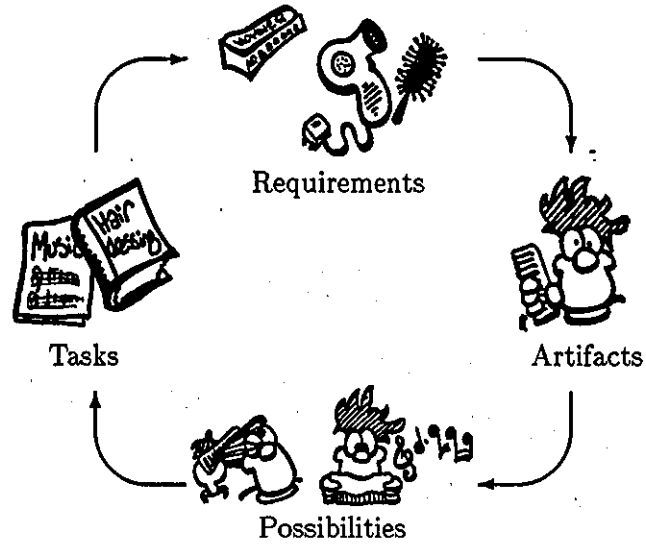


Figure 3-2: The task-artifact cycle

Carroll [26, 27, 31] however vigorously and persuasively defends the practice of of HCI as a craft. He rejects the applied science cycle (figure 3-1) on the grounds that HCI has 'infinite detail' — it is too complex for successful abstraction therefore the discovery mapping needs to be infinite and hence is worthless. Furthermore Carroll argues that what actually happens in HCI practice is that the applied science cycle works backwards; successful designs are developed and emulated then academia analyses those designs.

Carroll therefore proposes a replacement for the applied science cycle; the task-artifact cycle (see figure 3-2) in which artifacts (or tools) are taken as central to HCI practice.

In the diagram we show how a comb may develop through the task/artifact cycle; a comb is an artifact (or tool) and its use may suggest other possibilities about how it could be used. In this case we suggest it may be used for combing hair or it may be wrapped in paper and played as a musical instrument; the use of the artifact changes the task. We can then analyse these new tasks and suggest improvements to the artifacts; maybe a harmonica if we concentrate on the musical side or more sophisticated hair-dressing tools if we stick to the comb's original purpose.

What HCI practitioners do with tools is how HCI develops. Using a tool alters how the task itself is done and this may in turn alter how the tool is used; tools may then alter the task and new tools may be created to adapt to this new task. Therefore tasks and tools are dynamic

and the task-artifact cycle shows how they relate. Carroll's artifacts implicitly embody much of what is expected of explicit theory, however they may be too complex to reduce to explicit theory. Furthermore Carroll argues that his task-artifact cycle describes not only the operation of HCI practice but of most fields in which design takes place.

In summary Carroll asserts that what is important about HCI practice is its product — usable systems, rather than the methodology which brought these systems about. Carroll and Campbell [29] berate Newell and Card [87] for suggesting that scientific strictures be imposed on HCI practice. They argue that such strictures are limiting rather than enhancing to the field and accuse Newell and Card (and others of a theoretical bent) of trying to impose behaviourist principles on HCI where the emphasis is on *describing* human behaviour rather than *explaining* behaviour. Description was seen as inherently more 'scientific' than explanation, therefore attempts to understand what motivates behaviour were ignored in favour of mechanistic cataloging of behaviour. In their defence Newell and Card [88] responded that this was not what they meant at all and the debate disintegrates rather into disputing the semantics of what Newell and Card said or implied in their original position statement [87]. Interestingly from our point of view Carroll and Campbell dismiss any attempt at predicting behaviour as part of this neo-behaviourist approach.

3.2.3 The theory of HCI debate — some thoughts

No-one would deny that a considerable amount of HCI is, by its very nature, going to be experimentally based, yet an experimental approach can never *guarantee* success in building usable systems. Successful HCI should then be a combination of experimental and theoretical approaches; at an early stage experimental HCI pushing the boundaries and theoretical HCI organising and rationalising the results demonstrated experimentally. As the science base becomes more concrete and reliable it may push the boundaries itself. HCI theory cannot exist in isolation to the real world, but experimental HCI can exist without reference to any theory — the point is how far can HCI advance without sound theory and how much impact is it likely to have on such areas as software engineering if it is always to be seen as a 'woolly' craft. Carroll and Campbell argue that the 'imposition' of scientific 'strictures' on HCI will be detrimental to HCI practice. 'Imposition' and 'strictures' are rather emotive terms; how different the argu-

ment sounds if we discuss whether HCI should be complemented by scientific principles. Carroll and Campbell's underlying implication is that theoretical approaches are somehow barren of creativity and imagination. 'HCI [is a] distinct sort of science: not a mechanical application of academic psychology' [30] but there is no reason why a designer cannot use insight and original thinking in both an experimental and a theoretic setting.

The compiling of theories has advantages, not only in some undefined future, but also in the present. Long and Whitefield [81] show how some rather different pieces of human factors work can be usefully brought together under a single framework such that the whole is greater than the sum of its parts. When it is unclear (as it often is) where certain pieces of HCI work should sit in the grand scheme of things the use of such frameworks can give extra, invaluable context to the work.

Much of the debate ranges over the use of purely psychological theory in HCI, but this is to state less than half of the problem area. HCI needs not only to draw on a much more powerful and large scale psychological theory [30] than is currently available, but also computational and other relevant theories (such as sociology). Viewing HCI as a sub-set of psychology is to greatly over-simplify the problem and shows that HCI still has some way to go to becoming genuinely inter-disciplinary.

Carroll argues that his artifacts implicitly fulfill the role of explicit theory and it is unclear whether explicit theory may at some time replace the implicit understanding of artifacts. Yet explicit theory is much preferable to implicit theory. Even if not all implicit understandings can be formulated into explicit theory this is no argument for failing to make as much explicit as possible. Furthermore Carroll backs his arguments on the basis that small details may have large effects. If so, surely such a detail is not, in usability terms, small.

Without wishing to be too controversial one may wonder how motivated the system design community (both academic and commercial) is to produce usable systems. Commercially speaking, how important *is* usability? (See [115, 19] for a thought provoking discussion on 'bugs', their propagation and users' acceptance of them.) Computer system design suffers (or benefits, depending on your level of cynicism) from being able to market sub-standard products without considerable outcry from users. If one was to purchase a less than usable spanner one would return it to the shop. Yet users faced with less than usable software packages are more likely to

attribute the usability problems to their own technical incompetence than to the incompetence of the designer in building a usable system. The reasons for this may warrant a field of socio-psychology in its own right, yet this lack of necessity for guarantees of usability may go a long way to explain the lack of theory.

For example, aeronautics now has a considerable body of predictive and explanatory theory associated with it and no-one would deny that the task of an aeronautic engineer, that of getting several hundred tons of metal safely and controllably into the air, whilst minimising noise and fuel consumption, maximising comfort to the passengers and a hundred other considerations is hugely complicated. Although in the early days of aviation planes were built using the sort of methodology advocated by Carroll, they are not now, mostly because in the early days when flying carried a high risk it was limited to those pioneers who knew and accepted that risk, whereas nowadays the end-users are everyday people who would not expect there to be a significant risk in flying. Purely experimental approaches do not guarantee low levels of risk and building aeroplanes on a purely experimental basis would (and did) have an unacceptable cost in crashed aeroplanes and loss of human life. The flaw in Carroll's argument is that the end-user population of an area of endeavour characterised by his task-artifact cycle tends to be small and complicit in its risks. Computer users can no longer be assumed to be pioneers of the technological frontiers, and inflicting unusable software on them is quite simply unfair. Experimentation is perfectly justified, but not if that experimentation is performed (as it so often is) on unsuspecting and undeserving end-users. Lastly, the problem compounds itself; as users are presented with more and more unusable systems the impression that unusable systems are unavoidable will become more widespread.

So we argue for a middle ground; not denigrating experimental approaches to HCI, or ignoring the contribution theoretical frameworks can make to the field, or (significantly) overstating the impact theoretical approaches may have. A theoretical framework should not be 'imposed' on HCI researchers and few (if any) writers would suggest such an imposition. However, as Carroll states, HCI research is moving at considerable speed; so fast that it is unlikely that theoretical approaches can keep up. But this does not render such approaches useless; indeed if masses of data are being collected and presented in the HCI literature then the need for a rigorous framework to compare, contrast, validate or otherwise analyse such data becomes more and

more necessary.

3.3 HCI theories and frameworks

Several writers have suggested frameworks suitable for formulating principles of HCI knowledge. Once such a framework contains HCI knowledge, then we could describe this as an HCI theory. As we discussed previously there is a long way to go before such a state of affairs is reached, but the laying down of such frameworks is a good first step.

We shall review the work of Norman [89], Long [80] and Storrs [110]. Norman's is the most influential, Long's shows how HCI can be part of an engineering discipline and Storrs' is (we believe) the most comprehensive and wide ranging.

3.3.1 Cognitive engineering

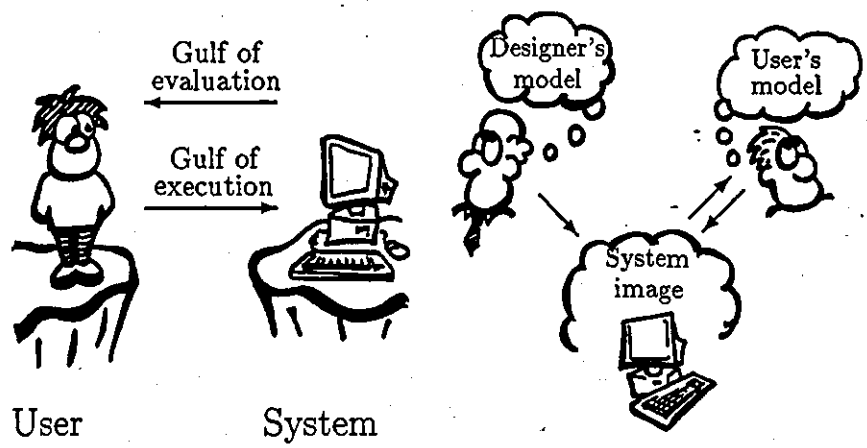
Cognitive engineering is the name Norman [89] gives to the endeavour the goals of which are...

1. to understand the principles underlying human actions that are relevant to designing usable computer systems, and

2. to devise systems that are pleasant to use. Note the emphasis here is on pleasure of use, not ease of use or ease of learning.

Norman offers a theory of actions which enumerates several processes which the user must go through in order to interact with a computer. These actions lie on his model of the 'gulfs' between a user and a computer (see figure 3-3(a)). There are two gulfs; the gulf of execution, which relates to how difficult it is for the user to pass intended information to the computer, and the gulf of evaluation, which relates to how difficult it is for the user to understand what the computer is trying to tell him. Obviously execution is concerned with user articulation and input to computers and evaluation is concerned with user perception and computer output. The narrower these gulfs the less effort the user has to expend to interact with the computer and therefore the more pleasant the user finds working with the computer.

Norman also describes what he describes as 'conceptual models' which are essentially the beliefs the user holds about the system. Users construct mental models of computer systems



(a) The gulfs between a user and a computer

(b) The conceptual models of an interactive system

Figure 3-3: Norman's models of interactive systems

from their interactions with them and the documentation provided about them. Norman calls this the 'user's model'[1]. He shows how user's models are derived from the 'system image' — the appearance the system gives to the outside world and the 'design model' — what the designer wishes the user to believe about the system. (See figure 3-3(b).) A usable system is one in which the user is presented with the system image in a clear and consistent manner. Furthermore Norman states that it is easier to present such clear system images for systems with a specific usage rather than more general systems.

Norman's work is one of the better and most well respected attempts at setting a framework for HCI. It is, however, biased to the human side of human-computer interaction and gives little indication of how things work on the computer side. More tellingly Norman's framework is based around what is now an out-dated model of human-computer interaction. One human is visualised doing one task on one computer. Modern systems allow many users to co-operate in many simultaneous tasks using a variety of inter-networked computers. Norman's model simply cannot cope with this complexity.

[1]User's model, user model and models of user are all common in the literature and may have quite different meanings. Care should always be taken when coming across terms containing the words 'user' and 'model'.

3.3.2 Cognitive ergonomics

Long [80] attempts to define and (in many respects) formalise a discipline of 'cognitive ergonomics' which is a sub-discipline of HCI concentrating on the human factors side of things. Much of Long's work therefore concentrates on building a framework for such activities as providing better training material and help facilities for already existing computer systems. However, Long first lays down a framework for HCI into which cognitive ergonomics fits. It is this more general framework that we shall look at here.

Long states the 'HCI problem' as being the 'optimisation of interaction between humans and computers to perform work effectively.' To deal effectively with the problem Long divides his framework into 'conceptualisation', 'methods' and 'interaction development practice'.

Conceptualisation — the domain of the problem area. Long broadly divides it into work, the entities that perform work and the effectiveness with which entities perform work. These divisions are then sub-divided again to define and show (for example) how work is distributed through organisations, how costs of system production can be measured, how performance and quality can be measured, *etc.*

Methods — descriptions and specifications of how work may be carried out within tasks. Methods are measured against some performance criteria, and the result of a task is its outcome.

Interaction development practice — the processes undertaken in order to solve the HCI problem. Long gives this as 'setting' (the statement of tasks possible using a system), followed by specification, implementation and evaluation. In other words, the traditional system life-cycle, with a task-analysis tacked on the front.

Long characterises three types of support for interaction development practice — craft, engineering and science (see figure 3-4.)

Craft practitioners acquire and maintain knowledge about interaction development informally. Being informal it cannot be applied generally or validated. Long asserts that this is how most HCI practice is currently carried out.

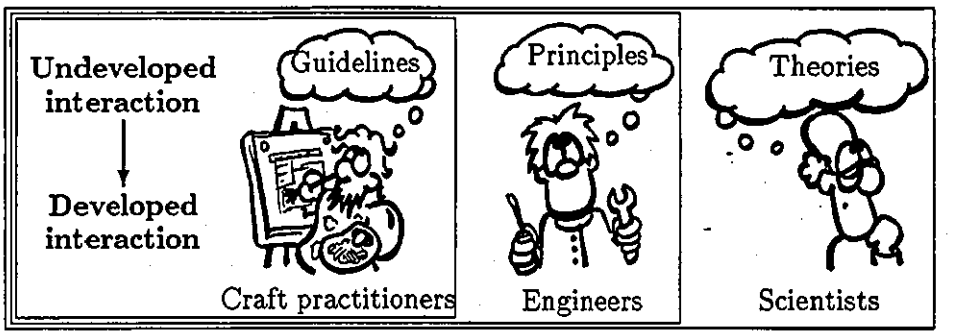


Figure 3-4: The types of support for interaction development practice

Engineers design and construct artifacts to fulfill some need. To do this they use principles related to the artifacts themselves and the methods used to produce them. The aim of engineering practice is to increase the number and generality of these principles.

Scientists develop theories to explain and predict the principles used by engineers. For science to be effective the theories it develops must hold true in the real world and they must explain the engineering principles.

3.3.3 Multi-party interaction

Storrs [108, 110] describes a theory as being constituted of three parts:

A conceptualisation — what the theory is about. A conceptualisation should be precise, so that we know exactly what we are talking about and, more importantly, what we are not talking about, and parsimonious, so we are not offered irrelevant and confusing unnecessary detail.

Facts — what the concepts are observed to do. In other words the raw data of a theory.

Laws — relationships between facts. If the laws of a theory are successfully predictive, explanatory and quantitative we have a good, useful theory.

Although not offering a full theory, Storrs proposes a conceptualisation of a theory of HCI. He states...

> Any interaction takes place through one or more interfaces and involves two or more participants who each have one or more purposes for the interaction.

The four main concepts here are participants, interactions, purposes and interfaces. The above statement describes how they relate.

A participant is either a user or a computer, although what is actually meant by computer is not clear cut. Storrs takes a very abstract view that it is a participant that has been constructed by other agents (an agent being anything with the power of agency). Participants may assume other roles such as ownership, representation and social groupings.

Interaction is the the exchange of information between participants where each has the purpose of using the exchange to alter the state of the other. Any information exchange is a dialogue and the unit of information is an utterance. Utterances may be individually addressed or generally broadcast. According to Storrs, when considering interaction the following must be taken into consideration: how synchronous are the interactions; whether the interactions are mediated, how much co-operation takes place between participants; and what costs are expended in the interaction.

An interface is a set of channels through which interaction takes place. A channel is a system for transforming information from some internal form to some external form or vice-versa.

The purpose of an interaction is to achieve state changes. To assist this purpose participants must make their utterances intelligible — the participants to which they are addressed must be able to perceive them. Furthermore the utterances must be comprehensible — the receiving participants must be able to make sense of them and they must be morphogenic — they must be effective at changing the state of the receiving participant.

Storrs analyses his conceptualisation by showing that Norman's [89] framework could be comfortably subsumed by his conceptualisation and that it can give new insights into HCI. Indeed [108] uses the conceptualisation to argue against the move toward separable user interfaces. He also discusses some of the omissions (such as a notion of task, although Storrs [111] recently defined a task in terms of his conceptualisation) from his model.

3.4 Summary

In this chapter we have looked at HCI in a very broad way. There have been telling arguments presented in the literature both for and against the sort of theoretical approach to HCI that we intend our work will be part of. The various arguments presented show (if nothing else) that there are still deep suspicions between the various methodological 'camps' in HCI. We have argued not that our theoretical and formal camp is 'better' than the experimental camp, but that the two should work together much more closely as both can benefit greatly from one another.

There is not a theory of HCI available to us, but there are conceptual frameworks designed to hold HCI knowledge as and when it is discovered. A framework filled with useful knowledge will be a useful theory. A test for the utility of a framework is that knowledge in a framework is greater than the same knowledge outside a framework. Long and Whitefield [81] have gone some way to showing that this is the case in their cognitive ergonomics framework and we have included a brief summary of their work to demonstrate this fact.

However Norman's and Storrs' work shows more relevance to our approach. Storrs' work supplies much that is lacking in Norman's framework, in particular it is not limited to one computer and one human and is not biased either to the human or computer side of things. But Norman's framework is more generally accepted in the HCI community. We will draw ideas from both frameworks.

Chapter 4

A review of previous work in the field

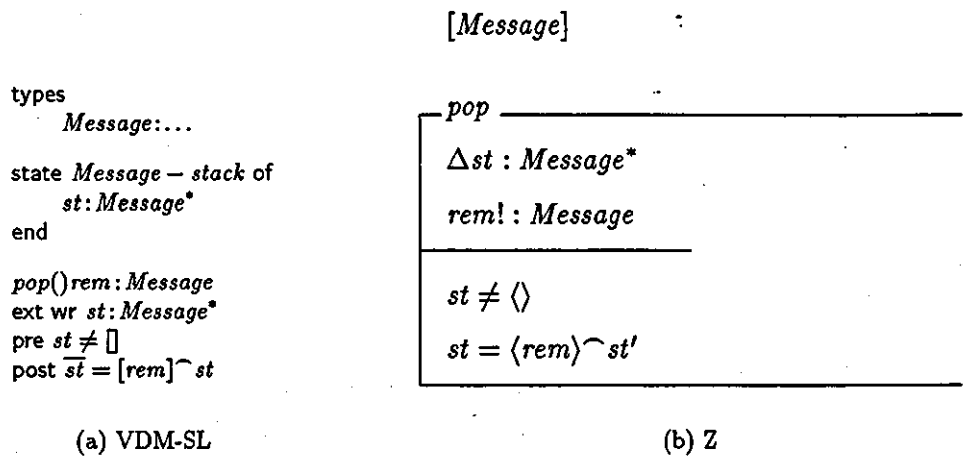
In this chapter we look at 'general purpose' formal specification techniques that have been used for describing a broad range of systems (section 4.1). The next three sections look at various formal models and techniques that have been used to describe interactive systems specifically. These three sections cover user models (section 4.2), interactive device models (section 4.3) and interaction or integration models (section 4.4). These sections broadly conform to the 'H', 'C' and 'I' of HCI.

4.1 Formal specification languages

4.1.1 Model based specification languages

We shall look at Z [107] and the Vienna Development Method (VDM) [70] as exemplars of model based specification languages. Both languages are based on the language of discrete mathematics and express the 'functionality' of systems. By functionality we mean that sets of functions are specified which express relationships between values of the system state. Model based languages can also express invariants about the system state — conditions that must hold throughout the runtime of the system.

The system state is modelled using set theoretic structures such as sets, relations and sequences. Invariants are expressed as first-order logical predicates on the state. Functions are described using 'pre' and 'post-conditions' which are also expressed as first order predicates. If the pre-condition for a function is true at the time the execution of the function starts then once



(a) VDM-SL (b) Z

Figure 4-1: An operation to pop messages from a stack specified in VDM-SL and Z

the function is completed the post condition will hold. In VDM-SL (the specification language of VDM) the pre and post conditions are described explicitly, whereas they are more implicit in a typical Z schema.

Figure 4-1 shows the same piece of functionality specified in Z and VDM-SL.

Refinement calculi

There are rules for both VDM functions and Z schemas which define how they can be rigorously refined to correct implementations. VDM in particular has a tried and tested refinement calculus associated with it. (It is, after all, a *development method*, rather than simply a specification language.)

A refinement calculus is a collection of transformations that can be performed on a specification which should result in a specification that is more concrete and easier to develop into a correct implementation. These transformations have formal proofs associated with them which the calculus effectively 'hides' from the software developer. The developer can then semi-mechanically apply these transformations without having to resort to the rather arduous task of a formal proof.

Even with such a refinement algebra the refinement process is still only semi-mechanical; there will still be several 'eureka steps' required of the developer. Given a specification *spec* a refinement algebra does not (automatically) suggest a more refined specification *ref*. Typically it is up to the developer to suggest the specification *ref* and use the refinement calculus to show that the step from *spec* to *ref* is a correct refinement step.

Object orientation

There have been several extensions and enhancements produced for both Z and VDM. Most notable amongst these additions has been the provision of tools to incorporate the 'object paradigm' (*e.g.* [52]). The object paradigm conceptualises a system as a collection of interacting objects each with 'attributes' (a model of their state) and 'methods' (operations that can be performed on the attributes). Objects are organised into hierarchies of 'classes' so that attributes and methods defined on objects high in the hierarchy are inherited by their sub-classes lower in the hierarchy.

The object paradigm is very popular in programming circles and has undoubtably improved programming methodology (not least because it encourages developers to think in a more abstract manner). Its benefit to abstract specification is less clear however; abstraction is (or should be) fundamental to specification and a methodology that encourages abstraction in abstract specifications seems rather tautologous. There is no generally accepted definition of what the object paradigm *is*, and hence confusion arises; there is much terminology associated with the paradigm which is vague and shifts meaning from one practitioner to another. Furthermore, although an object class hierarchy can be a very useful tool in system modelling, several approaches to the object paradigm allow multiple inheritance (the hierarchy is then a graph rather than a tree) and we believe the ambiguities introduced by such an approach far outweigh any advantages.

We will treat the object paradigm with circumspection; we intend to make no claims that any of our work is objected oriented, although it may be retrospectively argued that we have used some of the tools of the object paradigm in our work.

4.1.2 Process algebras

Model based specifications deal with the functionality of a system, whereas process algebras deal more with the 'behaviour' of a system. They show how various processes in a system evolve through time and interact (or interfere) with one another. The most common exemplars of process algebras are Communicating Sequential Processes (CSP) [64] and the Calculus for Concurrent Systems (CCS) [84].

The process algebra approach to specification rose from the confusion that is introduced into systems by concurrency (doing more than one thing at a time) and non-determinism (the ability to make seemingly random choices about what activity to perform next). There is no particular reason why a concurrent system should be non-deterministic or vice-versa, but concurrency and non-determinism were introduced to the computer science community at roughly the same time and so, for historical reasons, the two are often dealt with together.

Both CSP and CCS show how single threads of activity (processes) can be combined using various operators which model concurrency, interleaving and non-determinism. A system is therefore seen as a collection of sub-systems which interact with one another and their environment. Although CSP and CCS tend to be mentioned in the same breath when listing techniques for describing concurrency they have rather different semantic underpinnings (much more different than Z and VDM-SL, for example). Indeed CCS is really a class of models of concurrent systems whereas it could be argued that CSP is a single entity in that class. This difference gives the two approaches fairly distinct benefits and drawbacks; CCS is more general whereas CSP is more practical.

However neither approach has much equipment for dealing with state descriptions, and descriptions of life sized systems can be very unwieldy without the addition of other notations to provide state descriptions.

4.1.3 Net theoretic approaches

Net theory covers a collection of modelling techniques based on 'Petri Nets' which were first conceived of by Petri in the 1950's and that have been considerably investigated and extended since (see [95] for an overview).

A simple net is a graph of connected transitions and places. A place may contain several

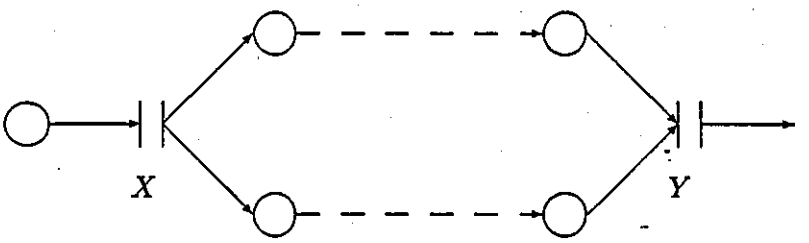


Figure 4-2: A simple Petri net showing a divergent and then convergent stream of activity

tokens. For a transition to occur there needs to be a given amount of tokens in its input places. Once there are the transition can 'fire' and pass tokens from its input to output places. The distribution of tokens throughout the graph determines the state of the system being modelled.

As the state description is distributed through the graph in this way, nets are useful for modelling distributed systems with concurrent sub-systems. Nets demonstrate well how transitions are dependent (or otherwise) on one another — they easily model systems where parallel, independent activities can take place in various sub-systems and what conditions need to be met when sub-systems need to co-operate.

As an example figure 4-2 shows a simple system whereby control diverges to two parallel, arbitrarily complex, independent streams of activity (denoted by the dashed lines) and then converges once both are complete. Transition Y cannot fire until both parallel streams are complete.

The formal basis of nets have been thoroughly investigated and there are several tools associated with them that can, for example, predict whether or not a modelled system is deadlock free, whether all states are reachable and whether the modelled system is live. Nets are, however, not Turing equivalent and therefore cannot be a complete specification tool; there exist systems that cannot be modelled adequately by nets. Nets are more expressive than finite state machines — a graphical modelling technique to which nets are often compared. As with all graphical notations nets have the benefit of not being initially frightening to non-formalists. However anything other than a very small scale example becomes very unwieldy. As is the case with process algebras it is not easy to describe complicated state models.

There have been many additions and accretions to simple nets in order to overcome their

deficiencies; there is work to object-orient them, to modularise nets in order to allow easier abstraction, to allow more expressive firing conditions on the transitions, to allow easier descriptions of complicated state models and so on. However many of these techniques tend to submerge the simplicity and elegance of simple nets.

Petri nets have been popular for modelling user-interfaces for interactive systems. We catalogue some of those approaches in section 4.3.6.

4.1.4 Modal logics

Several species of modal logics have been suggested for specifying computer systems. The most successful are probably temporal logics. Temporal logic was first proposed in system specification by Pnueli [97] and such logics have been taken up by several researchers since.

A simple temporal logic combines first order predicates along with temporal combinators $\Box$ and $\Diamond$. Given that P is a predicate the formula $\Box P$ reads 'henceforth P is true' and the formula $\Diamond P$ reads 'P becomes true at some time in the future.'

Temporal logics typically describe systems at a very abstract level in terms of 'safety' and 'liveness' properties. Safety determines what the system should never do and liveness determines what the system should eventually do. A system would be specified by...

$$sys \triangleq \Box \neg bad \wedge \Diamond good$$

...where *bad* is some condition we wish never to hold and *good* is some condition we wish the system to eventually attain.

This approach gives a good abstract approach to thinking about systems; we can think of a system repeatedly attaining the *good* condition whilst the 'path' it goes through to get to this condition is never '*bad*'. Alpern and Schnieder [6] showed that any system can be described by a conjunction of safety and liveness conditions. However they also showed that liveness properties may have some safety implicit in them. Hence it may not be possible to describe a system simply as a collection of properties which can be easily divided into safety and liveness. In more complicated systems notions of safety and liveness are likely to become inextricably intertwined.

Whereas model based specifications look at finite chunks of functionality, temporal logics

can describe infinite 'behaviours' of systems.

Maibaum [82] has used deontic logic to model system behaviour. Deontic logic uses operators that express obligation and permission on actions. Maibaum particularly uses this approach to make a more thorough distinction between 'pre-conditions' and 'pre-contexts' which are usually grouped together under the title 'pre-condition'. Deontic logic allows the specifier to distinguish between actions that *can* be performed once their pre-conditions are satisfied and that *must* be performed.

4.1.5 Discussion — notations for describing interactive systems?

The four groups of techniques and notations we have discussed here are all intended to apply to general purpose systems. It is argued (*e.g.* [2]) that none of these general purpose approaches and notations are well suited to the particular problems associated with interactive systems.

We believe the main problem is that interactive systems require both a thorough description of the system state *and* of concurrent behaviour. Therefore model based specifications and process algebras are individually inadequate for the job of describing interactive systems. Several of the approaches we shall discuss below bring a model based specification language and a process algebra together into a single language. This can make the notations rather unwieldy. Extended Petri nets can describe state and behaviour, but as we discussed they tend to be rather unwieldy.

We believe that recent work on temporal logic specifications has shown that both the state space and behaviour can be described in a single notation.

There are several other issues pertaining to the use of formal notations in specifying interactive systems we shall discuss more thoroughly towards the end of this chapter (section 4.5).

4.2 User models

In this section we look at formal and semi-formal user modelling techniques. The aim of these techniques is to provide predictions of how a user population will work with a given device. Hence an analyst can provide predictions about how a proposed device will be used and therefore how usable the device is. Such predictive analysis allows usability considerations to be included early in the system synthesis process. Such models also have the advantage that they present

user considerations in a language more easily understood by system designers. This is only a secondary (almost accidental) benefit however for most user models.

We do not intend our work to deal in depth with user modelling so only a brief overview is given here. We contend that cognitive science is not currently strong enough to support fully rigorous formal models of user cognition and intention. We do intend to show how and where the *results* of an user-modelling analysis (formal, semi-formal or not formal at all) feed into our interactive system synthesis process, so a catalogue of user models will be of use for this purpose.

4.2.1 Models based on the GOMS approach

GOMS [25] (Goals, operations, methods and selection) is a modelling technique that assumes the user has a hierarchy of goals. These goal hierarchies are built out of methods, which are algorithms for achieving goals, selections, which show how the user decides between methods and operations which are the atomic actions which the user can carry out. The granularity of the analysis is set by the operations which, depending on the analysis, can for example range from 'Edit document' to 'Press Enter'. Times are assigned to the operations at the 'keystroke level' and by analysing the goal hierarchy a rough prediction of performance can be obtained. Other potentially useful properties can be derived from a GOMS analysis — *e.g.* the depth of the hierarchy is claimed to be proportional to the demands placed on short-term memory.

GOMS is based on the 'model human processor', a very idealised cognitive model of a user as an information processor, which has been heavily criticised for the gross simplifications it makes.

CCT

CCT [72] (Cognitive Complexity Theory) combines a GOMS-like model of a user with a dialogue grammar in the form of a transition network. The two can be 'run' against each other to give a measure of 'cognitive dissonance' — the difference between what the user knows and what he needs to know. The dialogue grammar can be executed to give a rough dialogue prototype.

UAN

UAN [63] (User Action Notation) is a notation based upon GOMS but allows an analysis of up-to-date mouse driven windowed systems (which GOMS would have problems with.) UAN relates user actions to the interface feedback and the underlying interface state. There is no fixed level of abstraction — tasks can be built (using temporal relations) from sub-tasks in a hierarchical manner.

UAN was designed primarily as a communication tool. Task behaviour is described in a way that can be passed between designers and (hopefully) unambiguously understood. The notation is 'open' — it can be updated and extended according to the in-situ needs of designers. It has, however, unclear semantics and large problems can become cumbersome.

4.2.2 Models based on formal grammars

There have been several attempts to formalise user-interface 'consistency' using formal grammars. Consistency is held to be 'doing similar things in similar ways' [101].

An oft-quoted [104, 61] example of a highly consistent system is that of the Mac Paint package. The actions the user must perform to achieve two similar tasks — drawing a rectangle and drawing an ellipse — are compared.

To draw a rectangle Select the rectangle tool, place the mouse pointer at one corner of the desired rectangle, press the mouse button, drag the mouse to the opposite corner, release the mouse button.

To draw an ellipse Select the ellipse tool, place the mouse pointer at one corner of an imaginary rectangle that will bound the desired ellipse, press the mouse button, drag the mouse to the opposite corner, release the mouse button.

The actions are very similar — hence a consistent interface.

Reisner [100] uses BNF rules to describe dialogue grammars. The complexity of these grammars is shown to be inversely proportional to the ease of use which users report. This is a purely syntactic analysis, no account is taken of such things as familiarity with the dialogue.

TAG [94] (Task-Action Grammar) is a parameterised grammar which is used to analyse the syntax of interactions and hence give a measure of consistency and provide a tool for modelling

world knowledge.

Both Reisner's grammars and TAG are purely concerned with 'articulation' — how the user communicates commands to the computer (the gulf of execution in Norman's [89] terminology — see section 3.3.1). DTAG [65] attempts to rectify this and some of TAG's other notable deficiencies, but it is held [3] that such embellishments simply make an already cumbersome notation more so. Jacob's [66] state transition diagrams are similar to Reisner's grammars, but also allow for system output to be included amongst the diagram terminals,

4.2.3 PUM

PUM [118] (Programmable User Models) models the knowledge that the user needs to bring to an interactive system in order to attain task goals. A PUM analysis predicts user behaviour based on principles of rationality [85]. A PUM analyst proposes a model of user knowledge, a user task, a model of the computer device and a 'designer's intended procedure' which describes how the designer intends that the user will achieve a task with the device. From these models a prediction of the user's behaviour can be made.

Originally PUM models were directly (and rather inefficiently) encoded as production rules in the artificial intelligence architecture Soar [86] and so interactions could be automatically simulated. Later work [18] developed an instruction language in which PUM models were described and could be automatically translated into Soar productions. More recently [15] it has been shown that a PUM analysis can provide valuable insights into usability questions without the need to resort to 'running' models on cognitive architectures.

Currently [14] the PUM approach is being moved onto a more formal basis so that its insights can be captured and proved more rigorously.

4.2.4 ICS

ICS [8, 11] (Interacting Cognitive Subsystems) differs from the user models described so far in that ICS models are less about predictions of user behaviour and more about approximating the cognitive processes that underlie the behaviour. ICS has developed over a number of years by taking empirical evidence, attempting to explain it based on the ICS model and refining the model to better cope with the evidence. It models the transformations that take place on

cognitive information and how that information flows from one cognitive processor to the next.

ICS models the human cognitive processing unit as a composition of nine sub-systems, all similar in structure, but dedicated to individual cognitive processing tasks. For example there is a peripheral visual sub-system connected to the eyes which deals with hue, contour *etc.* and more central sub-systems which deal with implicational meanings and mental imagery *etc.* A sub-system receives flows of data which are copied to a store, transformed and passed on to other sub-systems. There are several overall constraints on the ICS model which determine its behaviour. Certain flows are not possible — for example visual information cannot flow directly to the semantic sub-system without first being processed by the object sub-system which processes the spatial information in visual input. All sub-systems continually generate information flows, but only some of these flows are stable and relevant to a task.

ICS models how cognitive resources interact and occasionally interfere with one another in the process of performing a task. ICS therefore gives a more 'holistic' view of cognition. ICS gives a good account of perception, a matter often overlooked by other cognitive models which deal more with user articulation. Also ICS has been used in conjunction with Interaction Framework [17] and Syndetics [47] to show how user models can be integrated with device models in formal frameworks. (See sections 4.4.1 and 4.4.2.)

EPIC

Executive Process-Interactive Control (EPIC) [71] is an implemented cognitive architecture based on general cognitive principles, but which has been applied to several HCI problems. EPIC models cognition by production rules and places this cognition within a strict context of perception and motor sub-systems. Cognition is therefore not considered separately to the motor systems, but necessarily interrelated. Performance measures can be strongly influenced by this interrelation.

EPIC also puts a lot of emphasis of its ability to model the user's cognitive processes acting concurrently. Users can overlap tasks (in a constrained manner) and EPIC has been used to predict that alterations to interface can make better use of this constrained concurrency. Executive processes (that are normally modelled by their own separate sub-system) are simply modelled as one among many of the concurrent cognitive processes.

EPIC is a symbolic processor and therefore its analyses are fairly low level and performance based. However it has shown that its performance predictions are useful in an engineering context. ([71] summarises much of the practical work on interface design so far undertaken by EPIC researchers.)

4.3 Formal models of interactive systems

This section looks mostly at the 'York approach' to formalising HCI issues. Dix [41] describes the York approach as a formalisation of interactive systems 'from the user's point of view'. The York approach is about capturing the *whole* of an interactive system rather than just the user interface. There are several pieces of work that aim to model user interfaces and they are described towards the end of this section.

4.3.1 Usability properties

The York approach identifies models of interactive systems and 'usability properties', the fulfillment (or otherwise) of which is claimed to have relevance to the usability of a device. It can then be formally proved whether the interactive system model will fulfill these properties.

Typically such properties are claimed to be the *minimum* required of an interactive system. Interactive systems that fulfill usability properties are not guaranteed to be usable, but systems that do not fulfill them are guaranteed to be very *un*usable.

We would argue that usability properties are a good tool in the construction of a usable systems. It is unlikely that even a highly usable device will accurately fulfill all the usability properties that have been proposed in the literature. A synthesis process for interactive systems will do well to measure the system being constructed against the properties to spot potential problems. 'Predictability' (defined in the next sub-section) is a very strong requirement to make of a system. However a designer employing a formal approach will be able to show when the system is going to be unpredictable and, what is important, should explain *why*. In other words usability properties should be tools for exposing possible problems with a design and therefore making designers give explicit justification for designing a system that does not fulfill certain properties. Usability properties are not really about forcing designers to uphold (possibly very

restrictive) properties in their systems.

However it is not clear that the properties formally described *are* very relevant to users, and even if they are, it is not clear that the mathematical definitions given correspond to the psychological definitions of the properties. This is called the 'formality gap' [43]. Here we see the problems caused by the theory vacuum of HCI — ideally we would like a set of properties the fulfillment of which guarantees the usability of a system, but there is no such work available in the literature. Abowd *et al.* [4] have set a space for structuring such properties, but this space shows signs of being determined by what *can* be formalised rather than what *should* be.

4.3.2 PIE models

Of the several models produced by researchers at (and associated with) York University the most abstract and simplest is Dix's PIE model [44, 41]. The behaviour of a system is expressed as a function I between 'programs' (sequences of input tokens) P and the resultant output from an effect space E.

$$I : P \rightarrow E$$

Hence the output of an interactive system after a sequence of inputs p is $I(p)$.

There are several extensions to the PIE model, most notably the red-PIE model (see figure 4-3) whereby the effect space is divided into a result space R and display space D. There are two functions r and d which take the effect to results and displays respectively. Results are in some the way the targets of an interaction; for example a hard copy of a letter from a word-processor. Displays are more ephemeral, intermediate effects such as screen displays. This division is a useful and oft-used formal trick which allows a relationship between what you see and what you get to be captured. This, then, has obvious relevance to the so-called 'WYSIWYG' (What you see is what you get) class of interactive systems.

From even such a humble model Dix shows that he can formalise the following properties which have relevance to the user...

Predictability — can the user deduce from what has happened so far in the interaction, what the effect of further input will be?

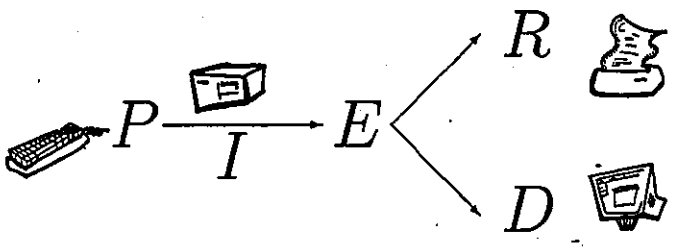


Figure 4-3: The red-PIE model

$$\forall p, q, r : P \bullet I(p) = I(q) \Rightarrow I(p^\frown r) = I(q^\frown r)$$

In words; 'if two sequences of input produce the same effect then any identical continuation of those inputs also produce the same effect.'

Observability — what clues does the external appearance of the system give the user about what is going on internally? Observability is based on the red-PIE model.

$$\forall p, q : P \bullet d(I(p)) = d(I(q)) \Rightarrow r(I(p)) = r(I(q))$$

In words; 'if two input sequences generate the same display then they also generate the same result.'

Reachability — is it possible to get from one state of a system to all others? Can the user get himself into a dead end?

$$\forall p, q : P \bullet \exists r : P \bullet I(p^\frown r) = I(q)$$

In words; 'after any sequence of inputs p there is a further sequence r which gets the system to the state generated by any sequence q.'

The other extensions to the PIE model are catalogued below. Most of them are described in [41], the references given below are where they were originally published.

Windowed systems [42] — several windows are modelled using a set of PIEs with a single PIE as a window manager. If each window represents a single task then the interference between each window should be minimised. Independence is formalised and several methods for capturing such independences are suggested.

Temporal models [39] — the effect of real time on usability is analysed and a temporal aspect is added to the PIE model to accommodate it. Keyboard buffering and display strategies are explored and a mechanism to ensure predictability even in systems with slow response times is suggested.

Non-determinism [40] — its is argued that non-determinism (or at least *apparent* non-determinism) can be generated at the interface by such things as window interference. Non-deterministic apparatus is added to the PIE model and several common interface problems are analysed.

The PIE model is 'unbalanced' in that it treats input as events — occurrences that have minimal duration such as key strokes whereas output is status information which has duration. Dix and Abowd [37, 5] argue that this is problematic. The mouse position, for example, is not naturally modelled by events. Trying to model parts of a system in way that is not natural to them can lead to problems...

- they are more likely to be specified incorrectly,
- the specification will be difficult to read, and
- the specification may very well be ignored.

Dix and Abowd argue that an unbalanced model will be difficult to modularise and decompose. Oddly interactors (see below) are held to be 'compositional but asymmetric' — why this is the case, why interactors are an exception to the rule, is not investigated. A model is proposed that is symmetric and can model both input and output as events or status.

State-display model

Another modelling technique developed at York which is a little more concrete than the PIE could be described as the 'state-display' model. The internal state of the system is given explicitly

(internal state is implicit in most formulations of PIE) and there are a set of commands for navigating amongst these states. States are further related to a set of perceivable displays.

Such a model was introduced in [60] to describe direct manipulation interfaces. The key idea is that the user edits the screen representation (rather than the internal state) and the internal state changes in response to what is displayed. Using the model it was shown that it is important to describe what remains constant in the system as well as what changes.

More recently [61] the state-display model has been used as the basic model from which more involved and less abstract models are derived. Properties originally formulated using the PIE model (observability and predictability) have been reformulated using the state-display model.

Yet another similar model; 'interactive processes' was proposed in [112]. CSP-like process algebra constructs were introduced and it is shown that trace histories of interactive processes should be constrained not only by internal state to state relationships but also by external trace constraints. The model shows many of the features of redPIE — states are divided into displays and results to formulate WYSIWYG-like properties. A large collection of other properties were formulated including most (if not all) of the properties described by PIE and its extensions. Interactive processes also had refinement rules defined on them so that more concrete specifications could be correctly developed from abstract ones.

4.3.3 The agent model

Abowd [2] took interactive processes and developed them into the agent model. The agent model allows the descriptions of interactive systems developed in the PIE model to be decomposed so that they can be related to more concrete, practical notations and techniques. The agent model represents a bridge between the very abstract models developed at York and architecture models (such as PAC (Presentation, abstraction and control) [33] and the Seeheim model [96].)

Abowd models a system as a collection of interacting agents which respond to and cause events. Like interactive processes agents' behaviours are defined by a mixture of state transitions and trace constraints. Much is made of the practical potential of the agent model; it is intended to be a tool for building interactive systems, rather than simply reasoning about them as is the case for much of the other York work. A 'user-friendly' language is given for describing agents by those not of a formal leaning.

4.3.4 Template abstractions

Template abstractions [62, 102] allow parts of a system model which are held to have psychological relevance to be abstracted from the whole. Hence just the psychologically interesting parts of a system model can be captured and analysed.

The properties defined on the template model are effectively a 'loosening' of the very strict properties defined in PIE-like models. The properties can then be applied to interactive system models in a more realistic, setting dependent way. However this 'loosening' is made in a way that is inspectable — the loosening is based on an abstraction that is both formal and is held (formally, semi-formally or heuristically) to have psychological relevance.

For example we can re-express the 'visibility' property based on a system model where S is the set of all states the system can assume and v is a function from states to displays.

$$\forall s, s' : S \bullet v(s) = v(s') \Rightarrow s = s'$$

This does not however express which parts of the state are relevant to the user. Two functions are defined on the state and display — a result template r that takes states to information that is task relevant, and a display template d that takes displays to pieces of information that the user can perceive and finds useful. Display templates are partial functions — the user may find no useful information on the display.

A system is held to be 'output correct' iff...

$$\forall s, s' : S \bullet \begin{pmatrix} d(v(s)) \text{ is defined} \wedge \\ d(v(s')) \text{ is defined} \end{pmatrix} \Rightarrow r(s) = r(s') \Leftrightarrow d(v(s)) = d(v(s'))$$

In words; 'two states that contain the same task relevant information are perceived to be the same.'

Templates allow for psychological information to be included at a level of formality that is sensible for the context. One of the problems with simple PIE models is that the properties it allows to be defined are very strict and it is difficult to justify them in psychological terms. Templates allow for a 'loosening' of the properties to an extent that is reasonable and justifiable on a psychological basis.

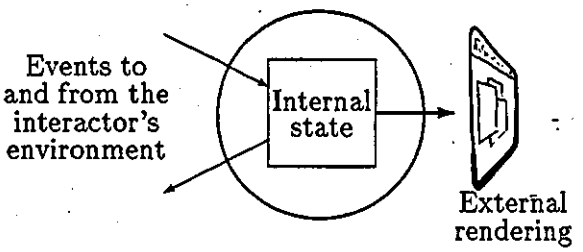


Figure 4-4: An interactor

4.3.5 Interactors

Interactors [48] are units of functionality in an interactive system. An interactor has an internal state which is 'rendered' to a some representation which is perceivable by the users. An interactor receives and transmits events to and from its environment which cause changes in its internal state. An interactor is depicted in figure 4-4.

Interactors are a concept intended to be abstract from any one particular formal notation. Most interactor work is presented in notations derived from and similar to Z, but there has also been work to show interactors in Modal Action Logic [103].

An interactor is characterised by a state that is internal to the interactor and a collection of rules describing how this state can be legally altered by events generated by other interactors and how those events can be generated. Furthermore there is a description of how the internal state is rendered to the user, usually by marking variables in the state as being visible or audible.

Duke and Harrison [48] show a 'tool kit' for formally building interactive system models based on interactors. They include such things as operators for describing synchronous communications between interactors and hiding events that are 'internal' to a composite of several interactors.

The interactor model is linked to an interaction model based on partial orders of events [51]. Such partial orders are held to be useful in capturing interaction dynamics where users have to perform certain actions, but in no particular order. Required interactions can be expressed as such partial orders and it can be shown whether interactive systems described in terms of interactors would produce those required interactions. Properties of interactions can be expressed in terms of both interaction models and interactors.

Both interactors and Abowd's agent model have been developed in the object paradigm tradition — they both consider systems and their user-interfaces to be strongly modular. Dearden and Harrison [35] argue that these approaches are more about describing interfaces in a rather concrete manner and have a 'limited range and therefore limited opportunities for re-use'.

However, interactors have been extensively used to model interactive systems in the Amodeus 2 project. Systems modelled include fairly 'standard' interactive systems and small scale interface 'widgets', but also very novel interfaces such as MATIS [92] (a multi-modal seat booking system for airlines) and AV [49] (an audio-visual communication medium). Much of the interactors work is summarised in the Amodeus Executive Summaries [50].

A related model, somewhat more operational, has been developed at CNUCE using the LOTOS formalism [54]. This imposes a structure on the internal state of an interactor based on a standardised model for computer graphics software. An abstract model of what the interactor represents is held (the 'collection') and this model can be passed to a 'feedback' section that generates the graphical representation of the interactor state. Input from the user is collected in the 'measure' which is then processed by the 'control' section into a representation more suitable for passing to the core functionality. Torres and Clares [116] have also worked on this model, concentrating more how the graphical representations can be formally captured.

4.3.6 User interface modelling techniques

Interactors are a step away from interactive system design into the more concrete area of user interface modelling. In the rest of this section we survey some other formal techniques for describing user interfaces.

Typically these techniques allow for a description of the interface and of the dialogue dynamics and then can either rapidly prototype the described interface so that it can tested against a user population, or it can be linked to a formal proof system to show that given properties hold.

This is not meant to be an exhaustive survey of such techniques — it is merely intended to give a 'flavour' of them.

Coloured petri nets for the OPADE system

De Carolis and de Rosis [34] used coloured petri nets to model the OPADE system. OPADE is a system which requires a highly adaptive interface because the user population is very wide ranging — they are of different professions and even of different native languages. These characteristics are captured in a 'user model'[1]. This 'user model' is then used to 'colour' the markers that pass around a model of the interface dialogue. From these models it is shown to be possible to capture whether it is possible to perform all needed tasks and a measure of cognitive complexity is based on the number of different transitions needed.

The ICO formalism

Palanque and Bastide [91, 90] presented the Interactive Cooperative Objects (ICO) formalism, which combined Petri nets with the object paradigm to model interactive systems. Their rationale for this combination is to produce a formalism that...

- captures both state and event information,
- describes both data and control structure,
- is modular,
- captures parallelism well, and
- is formal.

The device is structured into an architecture consisting of a functional core of non-interactive objects, a dialogue layer and a presentation layer. The functional core is structured according to the object paradigm and is linked to the dialogue layer which is captured in a Petri net.

A model expressed in the ICO formalised can then be verified against a collection of properties including deadlock absence, predictability, reinitialibility (the ability to get the system back to its initial state from any reachable state) and the availability of commands.

[1]Note that this is a different type of user model to the user models described in section 4.2 — this is a model of the user maintained by the device during run-time so that the interface can adapt to the user

TADEUS dialogue graphs

Elwert and Schlungbaum [53, 105] use coloured Petri nets to model interaction dialogues. These dialogue models are generated from task and user models that form a requirements analysis. The dialogue model can be used to automatically prototype a graphical user interface.

TADEUS (*Ta*sk based *De*velopment of *Us*er interface software) is a development process that is well supported by a collection of software tools that takes requirements definitions through dialogue graphs to interface prototypes. The interface prototyper embodies various interface design guidelines and heuristics. The evaluation of the interface comes (presumably) from testing the prototype on a user population — there is little evaluation done on the dialogue graphs themselves (other than the demonstration of freedom from deadlock).

Interface construction and verification

Bumbulis *et al.* [22] offer a technique for describing graphical user interfaces in a language based on Dijkstra's guarded command language [36] which then can in one direction be prototyped into a working interface based on various user interface toolkits. In a second direction the interface description can be recast in a more abstract model and then have various properties proved using such techniques as higher-order logic (HOL).

Specifying user interfaces in DisCo

DisCo is an executable specification language based on Lamport's TLA [74]. Systä [113] shows how it can be used to specify user interfaces at a high level of abstraction. Because DisCo is executable the specifications can be rapidly prototyped and an implemented animation tool is provided for this purpose, furthermore there is scope for formal verification of interface properties.

4.4 Interaction models and integration

Most of the work we have described so far concentrates either on the user or device side of an interactive system. For example the PIE model describes interactive devices and usability properties of those devices. However PIE has little linkage to a user model so that it is possible

to discuss in detail behaviour caused by differing user populations. Likewise many of the user models link to rather sparse and inexpressive models of devices.

In this section we consider two approaches that aim to bring both user and device models into a single framework, so that behaviour that results from the composition of user and device can be modelled and discussed.

4.4.1 Interaction framework

Interaction Framework (IF) [17] allows interactional trajectories and dynamics are formally modelled and then provides 'hooks' to user and device models. The motivation behind IF is that an analyst can propose desirable interactions and then propose user and device models which when put in conjunction will produce that desirable behaviour.

Several formal properties of interactions are defined. For example interaction framework allows the modelling of a task and the most ways of achieving that task in terms of 'interaction trajectories'. For any task there are 'canonical trajectories' which are the most efficient ways of achieving that task. Using these it is possible to define how much 'detouring' (*i.e.* inefficient interaction) there is in a given trajectory. Other matters that it is possible to capture in IF include 'user freedom' *i.e.* to what extent the user has the initiative in an interaction and to what extent the device places (possibly unnecessary) constraints on the order that the user issues commands to the device.

IF developed from a model of interaction that was abstracted from the agents that cause that interaction [10]. This model could then be hooked to existing user and device models and used to expose and validate the implicit assumptions that the user model makes about the device model and *vice versa*. More recent IF work widened the scope to include systems containing more than just two agents and there has been some work to show the movement from specification to implementation guided by the framework [16].

4.4.2 Syndetics

Whereas IF defines interactions and links them to differing user and device models, syndetics [46, 47] is an approach that integrates user and device models by expressing them in a single notation to create a unified 'syndetic' model. Having the whole system expressed in a single

notation considerably eases the reasoning that may be done with a model.

So far syndetic work has concentrated on expressing device models in an axiomatic style based on interactors and expressing a user model in a similar axiomatic style based on ICS. The notation used is MAL [103]. However syndetics is an approach that is not intended to be dependent on any particular device model, user model or notation. The axiomatic style allows for hypotheses to be proposed about the expected behaviour of the system and for these hypotheses to be formally proved.

The interactor style of specification is modified a little so that the rendering information (which parts of the internal interactor state are perceivable by the user) links directly to the peripheral sub-systems in ICS. For example a state marked as 'visible' in an interactor feeds into the visual sub-system, and a state marked as 'audible' feeds into the auditory sub-system *etc.* Once a device model and ICS model are combined it is then possible to show formally whether certain desired behaviours are going to cause undesirable effects to the user, such as unstable or incoherent data flows. If this is the case then it can be argued that the user is going to have difficulty performing the desired behaviour.

Syndetics is an attempt to bring together onto a single firm footing many of the design perspectives that are currently used in HCI. Because of the plethora of representations used there can be loss of information translating from one to the next if all the representations are not equally expressive.

We would argue however that the problem does not lie with the plethora of notations but the lack of rigorous underlying semantics for those notations. In a simplistic way we might assume that an interactive system model consists of device, user and behaviour models. Attempting to express all three in a single syndetic notation may place undesirable constraints on what models are expressed — a user model and a behaviour model are *very* different things and a unifying notation may not allow for the expression of features that are peculiar to each.

It is interesting to note the difference between syndetics and IF — namely that syndetics commits to a device and user model from the outset whereas IF starts with an interaction model and then hooks into device and user models later. A long term aim of the development of these integrational approaches should be to propose frameworks where different, large scale, questions can be asked with those frameworks, such as 'how do we get this behaviour?' or 'what is the

effect of changing this device model?' and so on. A good integration framework should allow the designer to start with any of the three entities, device, user or interaction as a fixed variable and then change other entities in the framework and assess the effect of this change.

4.5 Discussion

We have described several techniques for formally describing interactive systems and their user interfaces. The models that are expressed using these techniques can be evaluated in several ways — by rapid prototyping, or by proof of properties. Several of the techniques are more analytic than constructive, *i.e.* they are useful for analysing existing systems to describe why they are usable or not, but it would be difficult to construct systems with them.

It is worth emphasising that one of the main benefits of these approaches is not the actual evaluation of the proposed models, but the process of proposing the models in an abstract way. Inconsistencies and errors can be exposed by abstraction. In a similar way a benefit of getting a software engineer to propose a user model is that it gets the software engineer to think from the user's perspective.

There are some notable gaps in the literature which we describe below.

4.5.1 Case studies

Many of the approaches described have not been tested in an industrial setting and it is not at all clear that these formal approaches will scale up to real-life situations. It would be useful for there to be a collection of worked case studies that approaches could measure themselves against.

The range and complexity of the techniques we have surveyed can be quite bewildering. Much of the work in the literature says the same things, but in different ways. An accepted collection of case studies would help identify these commonalities as well as what is peculiar to particular approaches. This would allow a specifier to decide which technique suits their application best. Actually demonstrating that the techniques will scale up would also help dispel the suspicions of many 'mainstream' HCI workers who believe that formal approaches only work with toy examples.

4.5.2 Usability of notations?

It is often claimed that graphical notations aid comprehension, especially for those who are not formal experts. This is an important consideration for a cross-over discipline such as formal HCI. A problem with graphical notations is that they rapidly get intractable with complexity.

We would argue (though we have no particular evidence for our argument) that the problem may lie with the presentation of mathematical notations. A piece of formalism should be well laid out so that concepts that are closely allied to one another are grouped together. Furthermore the formalism should be accompanied by wholly adequate explanatory text. Too often formalism is presented in the literature with little accompanying explanation and readers are left to fend for themselves. Graphical notations force (to a certain extent) the grouping of closely allied concepts and hence this may lead to their reputation for being easier to understand, however a badly laid out diagram with no explanation would be much more difficult to understand than a well laid out and explained piece of textual notation.

There has been some work in investigating the usability of notations.

- Brun and Beaudouin-Lafon [21] include 'usability' amongst the twelve criteria by which they evaluate a taxonomy of formalisms. Unfortunately they do not describe how they assessed usability and furthermore describe good usability as 'when the description of simple things is simple and the description of complex things is possible'. This statement possibly confuses usability with how easy it is to modularise and abstract specifications. Perhaps a better definition would be 'when the description of simple things is simple and the breaking of complex things into simple things is simple'. Most of the notations evaluated (which range from GOMS, through petri-nets to Z) score rather badly on usability, and there seems to be no differentiation in usability for graphical and textual notations. In fact most of the graphical notations come off very badly in Brun and Beaudouin-Lafon's evaluation — they are criticised for being inexpressive and having poor modularity.

- Gray and Johnson [57] compare three notations (a temporal logic, Petri nets and XUAN [56]) with an emphasis on how well they express temporal properties. Essentially they conclude that each notation has its own strengths and weaknesses and that all three have significant usability problems.

- Johnson [67] found that users of formal notations expressed a strong preference for natural language descriptions but, apparently paradoxically, made less errors with a temporal logic notation that they expressed a distaste for.

So where does this leave us? Without a clear idea of which notations (if any) are usable, we argue that the important consideration is the underlying semantics. We contend that a well-defined semantics for a notation gives the models expressed in that notation some degree of portability. Co-operating workers could then express models in whatever notation suits them and pass them to other workers who could (in theory) re-express them in another more suitable notation. How easy this translation would be is another matter.

Because of the ambiguity in claims for notations' 'usability' it is unwise to make claims such as graphical notations being necessarily usable and clear.

Chapter 5

Introducing a Reactive System Specification Language (RSSL)

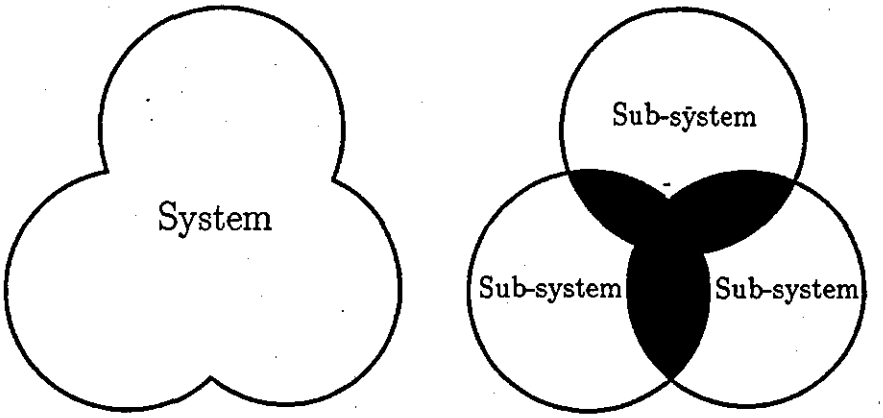
We view an interactive system as a specialisation of a reactive system. In this chapter we make clear our definition of a reactive system and propose a process for synthesizing such systems. We then introduce a Reactive System Specification Language (RSSL) for formally describing reactive systems. We introduce the language using a simple example and then by adding layers of complexity to it. We conclude with some suggestions about strategies for implementing systems described by RSSL, and making clear how RSSL specifications fit into a reactive system synthesis process.

5.1 Some introductory definitions for systems

We will begin by defining some terminology.

5.1.1 Closed systems

We intend to deal with the description and synthesis of 'closed' systems. A closed system is one that is entirely self-contained; it is not effected by any external agents and has no effect on external agents. If we were describing an interactive system consisting of a user and a computer then *all* we can describe is the interactions between the two. If we need to include other sub-systems (for example if the user were to set the clock on the computer by looking at



(a) A closed system (b) Open sub-systems

Figure 5-1: Splitting a closed system to describe its behaviour

his watch) then we must include these sub-systems in the description. Furthermore we assume a closed system to be sparse; it only contains those sub-systems that have an effect on the overall behaviour. Therefore we would *only* include a description of the user's watch in an interactive system if the user was going to use it in a way that would effect his interactions with the computer.

Note that when we refer to a 'system' we refer to the entirety of a system, often in the literature 'system' refers only to the computerised or automated part of a system. This is not our view of a system, when we refer to the computerised part of a system we call it the 'machine' or 'device'.

By it's nature a closed system has no externally perceivable behaviour. We therefore describe the behaviour of a closed system by splitting it into a collection of sub-systems and describing the system behaviour as how those sub-systems react to one another. See figure 5-1. When we split a closed system like this each of the constituent sub-systems must be 'open' to at least one other sub-system. If two sub-systems are open to one another we mean that they can in some way effect or be effected by one another.

Typically sub-systems that are open to one another will share some state space between them which is directly manipulatable and perceivable to each of the sub-systems. If we split a

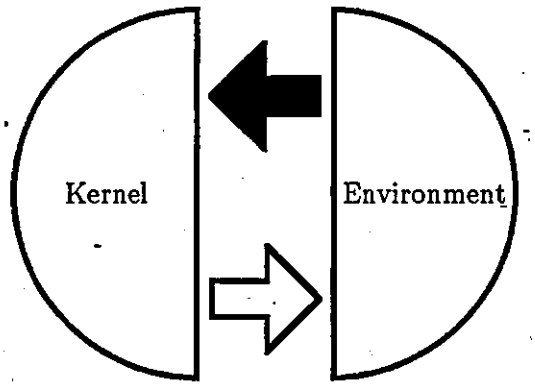


Figure 5-2: A reactive system

closed system into a collection of open sub-systems then the behaviour of the closed system is the changes that take place in the intersection of the sub-systems' state space.

In figure 5-1 we have split a closed system into three sub-systems. Each circle represents a sub-system and where two sub-systems overlap represents the state space shared between them. The shaded area shows the state space in which the the behaviour of the system takes place.

Note that this description of systems is recursive; each sub-system can be further split into other sub-systems in the same way as we have described the splitting of a single system.

5.1.2 Reactive systems

A reactive system is one in which the sub-systems into which it is split can be placed in two (exclusive) categories; 'kernel' sub-systems and 'environment' sub-systems. (The kernel sub-systems are usually referred to simply as 'the kernel' and likewise the environment sub-systems are simply referred to as 'the environment'.)

Assume that sub-systems use their shared state space to pass requests to one another. One sub-system does something to the state space that the other system can interpret as a request to perform some action.

The kernel 'reacts' to the environment but the environment does *not necessarily* react to the kernel. When the environment makes a request of the kernel then the kernel should respond to that request. The kernel can make requests of the environment but a response cannot be

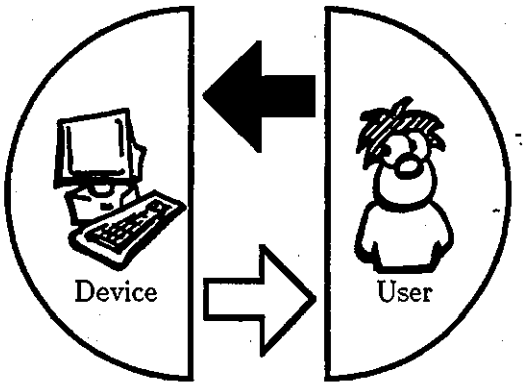


Figure 5-3: An interactive system

guaranteed. A reactive system is one that is 'unbalanced' in this way; there is an obligation on the kernel to 'do what it is told' but no complementary obligation on the environment. See figure 5-2; the solid arrow denotes the obligatory nature of requests passed to the kernel, the empty arrow denotes the weaker non-obligatory nature of requests passing the other way.

Interactive systems as reactive systems

Typically in the literature reactive systems are small scale and based on embedded hardware systems, such as lift or thermostatic controls. Following [113, 83] we define interactive systems to be specialisations of reactive systems where the environment is the user population and kernel is the computer device. (See figure 5-3.)

Typically the user will issue requests to the device by manipulating the keyboard or mouse (or joystick, trackpad, voice recognition unit, touch screen *etc.*) which the device must respond to (if it can). The device can issue suggestions and prompts to the user via the display (or whatever) but there is no guarantee that the user will respond to them. There *are* computer systems that demand responses from users. (For example, before hard drives were common Macintosh users were often asked to insert a system disk without, as is usual in the Macintosh interface, being offered a 'Cancel' option. Even this is not strictly obligating the user however, he may give up or simply switch the computer off.) However we believe these not to be typical interactive systems and a designer should come up with a good justification for designing a

system that relies on obligations on the user.

Note that our definition of an interactive system is not limited to one user and one piece of computer machinery; we can characterise the environment as one or a set of users and likewise the kernel as several computers.

5.2 A formal design process for reactive systems

A formal design process starts by describing the a system in very abstract terms and then adding layers of detail to that description until it exactly describes an actual working system.

5.2.1 Abstract descriptions and refinement

An abstract description is one in which many details that are considered to be irrelevant are deliberately omitted so as not to clutter the 'essence' of the system being described. Therefore an abstract description yields a wide set of possibilities for a system. The process of adding layers of complexity (known as the 'refinement process') narrows this set of possibilities by being more explicit about the system. If a system description yields a set of possibilities then a refinement of that description yields a sub-set of the original possibilities.

Take a trivial example; we want a system that repeatedly increases the value of the variable x. An abstract description of this system may say something like 'for every time there is some future time when x has increased.' This description gives a wide set of possible systems; so long as x gets increased we do not care how it happens. A refinement of this system is a system that repeatedly adds one or two to x. This refined system still provides a large set of possibilities (as we have not said anything about how soon one or two get added to x) yet it yields a strict sub-set of the possibilities yielded by the more abstract description which allows *any* number to be added to x. However, we need to be careful; refinement is not just a case of throwing possibilities away — if this were the case then the empty set of possibilities would refine all systems. Some of those possibilities may be crucial to the system; refinement is about reducing the possibilities for the system whilst still ensuring it does everything required of it.

5.2.2 Requirements

We propose that the most abstract description of a system is its 'requirements'. Requirements are a certain style of system description that concentrate on describing the problem we need a system to overcome, rather than worrying about the system itself.

A statement of requirements describes the behaviour of the shared state space (the shaded area in figure 5-1(b)) with no reference to the 'internal' behaviour of the sub-systems. Later on we shall discuss 'specifications' which are abstract descriptions of the sub-systems themselves. It is crucial to realise that fundamentally requirements and specifications are the same thing; they are descriptions of behaviour, they are however expressed in different styles. Pnueli [98, Section 1] compares and contrasts these two styles of system description. The notation we use for expressing requirements is that of a simple temporal logic, being a predicate logic (which we assume the reader to be familiar with) augmented with several operators that allow us to describe 'when' certain conditions are true relative to 'now'.

The state space

Recall from the previous section that we proposed that the behaviour of a closed system was the behaviour of the shared state space of its sub-systems. The first step in describing the requirements for a system is then to describe this state space which we do by listing the names and corresponding types of all the variables in this space. A variable is some entity that can change its value (or have its value changed) and its type[1] is the set of all values it can legally assume. The state space is considered (ultimately) to be discrete; variables in the state space assume exactly one value at a time; they cannot be thought of as being 'in between' some values.

Having described the shared state space we now describe the legal ways in which the values of that state space can change. The style we propose for the description of requirements is that of 'safety and liveness' properties first proposed in [73] and elegantly formalised in [6].

[1]Note that even though our approach is based on Lamport's TLA [74], we do not share Lamport's view that types are detrimental to a design process; we believe them to be valuable descriptive tools.

Safety

A safety property expresses that something 'bad' should never happen to a system. The temporal operator $\Box$ reads 'henceforth' or 'always', hence safety properties typically have the form...

$$\Box(\neg\text{something bad})$$

In words; 'it is always (at all times) the case that something bad does not happen.'

Liveness

Liveness properties express that eventually something good must happen. The temporal operator $\Diamond$ is read 'eventually' and so liveness properties typically have the form...

$$\Diamond(\text{something good})$$

In words; 'there is at least one time in the future when something good happens.'

Alpern and Schneider [6] showed that any property can be expressed as a conjunction of safety and liveness properties.

Some caveats

However, things are not this simple. Alpern and Schneider also showed that safety properties can have liveness implicit in them and *vice versa*, therefore some requirements may not be so simple and clean as a safety and liveness property with no interdependencies between them. Clarity of the description is the crucial point and it is more important to describe requirements in a way that is clear and elegant rather than getting too concerned about rigorously slotting the descriptions into safety and liveness boxes.

Furthermore liveness in a reactive system may become quite involved. Liveness in a reactive system is dependent on the behaviour of the environment; only the kernel needs to be 'live'; once the environment requests some action from the kernel then the kernel must eventually do 'something good' and produce a response. However the environment need not be live in that it may never make a request. Consider a drinks dispenser; when the customer (the environment)

puts sufficient money in the machine (the kernel) it must respond with a can of drink; it must be live. However, we cannot dictate liveness on the environment; the user may never put any money in the machine. Also kernels tend to be always live; a drinks dispenser that produces one drink then stops is not typical. Therefore liveness for reactive systems will typically have the following form...

$$\Box(\text{environment request} \Rightarrow \Diamond(\text{kernel does something good}))$$

In words; 'it is always the case that the kernel eventually responds to an environment request.'

This unfortunately blurs the distinction between requirements and specifications. To talk about the kernel and environment means we have made some decisions about the structure of the system and there is some system biasing in the requirements. It is rather utopian to think that we can cleanly slice requirements away from any idea of the system which fulfills those requirements and the biasing we have described here is minimal, so we do not consider it to be a problem.

We have shown several caveats to the simple idea of producing requirements as the problem we need a system to overcome in terms of safety and liveness properties. That said, thinking of a the problem space in terms of safety and liveness is a good first step in describing a system. It should also be noted that many formalists mistakenly omit liveness, assuming it to be somehow implicit in the specification. We do not assume that a system will do something good just because it *can*, we need to explicitly state that it *will*.

5.2.3 Assumptions and specifications

Requirements describe the behaviour of the shared state space (the shaded area in figure 5-1(b)). Assumptions and specifications describe the behaviour of each of the sub-systems such that when all the sub-systems are placed in conjunction with each other the shared state space behaves so as to satisfy the requirements.

Each of the sub-systems will either be already extant; in which case we make 'assumptions' about it, or it will need to be constructed, in which case we 'specify' it. Again assumptions and specifications are fundamentally the same thing; they describe the behaviour of sub-systems. We

draw a distinction so that we know which part of the system to concentrate on in synthesizing a system.

In slogan terms the relationship between requirements, assumptions and specifications is as follows...

$$\text{requirements} \triangleright \text{assumptions} \wedge \text{specification} \tag{5.1}$$

In words; 'the conjunction of the assumptions and specification should be a valid refinement of the requirements ($\triangleright$ denotes refinement) where 'refinement' adds detail to a description while still ensuring that the refined system does nothing illegal and does everything it must.'

Simplistic though it is formula 5.1 is one to which we shall repeatedly turn in this thesis. We assume that the requirements and assumptions have been defined and we take the point of view of a system designer whose job it is to specify, design and build the rest of the system such that formula 5.1 holds.

'Pnueli style' specifications

We base our specifications on the 'Pnueli style' [97] of specification. A specification consists of an initial predicate that describes all the states the system can legally start in and defines a disjunctive set of actions which describe how the system state can develop. Fairness conditions are also included to ensure that undesirable effects such as permanent lock-out do not occur. An action is a relationship between two states and describes a single change in system state.

Recall the very simple example from the beginning of this section; the refined system repeatedly adds one or two to the the variable x. Assuming the system starts with x at zero and we wish there to be a fair mixture of ones or twos added to x then we would specify the system as follows in the Pnueli style...

$$x = 0 \wedge \Box(addOne \vee addTwo) \wedge fair(addOne \vee addTwo)$$

$x = 0$ is unguarded by a temporal operator and is therefore assumed to hold at time zero. *addOne* and *addTwo* are actions that add one or two to x respectively. In order to express the relationship between two states actions are expressed using the variable decoration $'$ to denote

the value of a variable in the end state of the action. Undecorated variables denote the value of the variable in the start state of the action. Hence we could define *addOne* and *addTwo* as follows...

$$addOne \mathrel{\hat{=}} x' = x + 1$$
$$addTwo \mathrel{\hat{=}} x' = x + 2$$

We believe the Pnueli style of specification gives a clear and concise description of a system, what is more the specification is expressed in a temporal logic (which is a little more complicated than that needed for requirements) and hence the step from requirements to specifications is greatly simplified. Traditionally requirements would be expressed in temporal logics and specifications in abstract programs or automata (*e.g.* CSP [64], VDM [70] or Z [107] *etc.*) and expressing relationships between the two would be rather arduous. Expressing both requirements and specifications in temporal logics considerably eases the workload in showing consistency between the two.

Enabling conditions for actions and deontic logic

The two actions *addOne* and *addTwo* defined above are permanently enabled; they can always occur. However in most situations there will be certain conditions in which we do not wish actions to occur. For example we may wish to have an action that adds two to x only when x is even.

$$addTwoEven \mathrel{\hat{=}} isEven(x) \wedge x' = x + 2$$

$isEven(x)$ is the enabling condition for this action.

Maibaum [82] argues that it is not clear in most specifications whether the fulfillment of the enabling condition implies that an action *must* or *may* occur. Maibaum therefore uses the deontic logic of obligation and permission to make the distinction explicit.

Maibaum's definition of obligation is that if action A is obligated then it is permissible (it can occur) and no other actions are permissible. This is rather a strong notion of obligation and can cause problems for the unwary specifier, not least because having two actions obligated

simultaneously is self-contradictory. We will make use of Maibaum's ideas although our notion of obligation will be rather weaker.

5.2.4 How reactive systems fit in

We have introduced several concepts here, so let us see how they fit in with our ideas about reactive systems.

Typically we will make assumptions about a reactive system's environment and specify the kernel. In the case of interactive systems (where users are the environment) we cannot build the environment; we make assumptions about how it will behave and then build computer devices to work with it to fulfill the requirements. It could be argued that we could 'build' the users' behaviour by the use of user training and support material, but that is beyond the scope of this work. Our interest in this work is building computer devices that work well with given user populations, not modifying the user population so that it works well with a given computer device.

The kernel offers a collection of actions to the environment and the environment may request some of those actions to occur. The environment makes that request by causing the enabling condition of the action to be fulfilled. Once the enabling condition of a kernel action is fulfilled the action should (if possible) occur; the actions performed by the kernel have an obligatory nature. We describe the actions performed by the environment as being merely optional; even if they are enabled there is no guarantee that they will happen. A drinks machine will always be enabled to accept coins, but this is no guarantee that users will actually feed money in. Our specification therefore consists of an initial predicate, a collection of actions the environment may optionally perform and a collection of actions that the kernel must perform once enabled. Our notion of obligation subsumes the notion of fairness and therefore explicitly expressing the fairness condition is not required.

Having covered the grounding and context for RSSL now let us move on to the actual notations of RSSL. So as not to swamp the reader in a mass of new notation we introduce RSSL 'bit by bit' by starting off with very simple requirements for a queue, after which we develop the specification of the queue system. We then proceed to add further layers of complexity to the example and introduce new notation only when needed. It is worth pointing out that we do not

simply move through the design process in a linear way, we cycle and backtrack in the process, as happens in real life.

5.3 An informal description of a queue system

For the queue system there are input and output lists. Items should be taken from the input list, processed and passed to the output list. To keep in line with conventional list processing, items are taken from the head of the input list and added to the end of the output list. However, once removed from the input list, items do not need to be *immediately* passed to the output list so long as they get there *eventually.*

Initially we do not worry about the effect of processing individual items and just concern ourselves with the movement of items. We are looking at a closed space, so ultimately items are not added or removed — they simply move from one list to another.

There is a desirable end state that we wish to be attained, namely that all the items intially in the input list finish up in the output list and their ordering is preserved. In other words the value of the output list is eventually the same as the initial value of the input list. This is the liveness condition.

To complete the picture we also need to consider legal ways that items move from the input to output list. Once in the output list an item should stay there and once an item has been removed from the input list it should not return to the input list. Hence the input list should not get larger at any time, nor should the output list get smaller. This is the safety condition.

Effectively what has been described so far is a queue *server* rather than a whole queue system. By adding a definition of the environment (section 5.6.3) we describe the whole queue system.

This description fits in with the guidelines for describing requirements discussed in section 5.2.2 — we have described the state space, a liveness condition and a safety condition.

The state space — two lists of items, an input and output list.

A liveness condition — eventually the input list is empty and the output list has the same value that the input list had initially.

A safety condition — items can only be removed from the input list and added to the output

list.

As we discussed earlier in more complicated examples it may not be so easy to draw such an easy distinction between the conditions. Furthermore in reactive systems there tends to be no 'end' state — the kernel is in (possibly) non-terminating reaction to the environment. However, as a starting point for an analysis, it is always useful to think of requirements in terms of something good eventually happening and nothing bad ever happening.

Having described the requirements informally we can now begin to look at things more formally.

5.4 Formalizing the requirements and requirements engineering

In this section we shall first of all formalise the problem space previously described informally. We can then analyse this problem space and see if we can come up with a 'better' model. We argue that the problem space we have described is a particular instance of a more general class of problems and it is preferable to describe this more general class, it being more abstract.

5.4.1 Formalising the requirements

The state space

We have a set of items and two lists of items.

$$Item \triangleq \ldots \tag{5.2}$$
$$in : Item^* \tag{5.3}$$
$$out : Item^* \tag{5.4}$$

...where *in* denotes the input list and *out* denotes the output list.

The liveness condition

The liveness condition describes a relation between the start and end configuration of the state space. Namely that initially *in* is equal to some value denoted by x and *out* is empty. The end

configuration is where *out* has the value x and *in* is empty.

Formally...

$$live \triangleq \exists x \bullet \begin{pmatrix} in = x \\ \wedge \\ out = \langle\rangle \end{pmatrix} \wedge \Diamond \begin{pmatrix} in = \langle\rangle \\ \wedge \\ out = x \end{pmatrix} \tag{5.5}$$

In words; 'initially *in* has the value x and *out* is empty (recall that in a temporal logic formula, sub-formulae that are not guarded by a temporal logic operator are understood to hold initially) and eventually *out* has the value x and *in* is empty.'

The safety condition

The safety condition describes the 'boundary' of legal configurations of the state space. In this case we only want items to move out of the input list and into the output list.

Assume that *suff* and *pref* are functions that return the set of all suffixes and prefixes to a given sequence respectively (see B.8 and B.7 in the appendix). We can give the safety condition by describing the input list as always being the suffix of its earlier values and the output list as always being the prefix of its later values.

$$\begin{aligned} safe \triangleq \Box(\exists a, b \bullet \\ in = a \wedge \Box(in \in suff(a)) \wedge \\ out = b \wedge \Box(b \in pref(out))) \end{aligned} \tag{5.6}$$

In words; 'it is always the case that a and b are the current values of *in* and *out* and all future values of *in* are suffixes of a and b is always a prefix of future values of *out*'.

The requirements

The requirements for the queuing system are a conjunction of the safety and liveness conditions.

$$reqs \triangleq live \wedge safe \tag{5.7}$$

Even in this simple example it can be shown that the liveness and safety conditions are interdependent on one another in defining the required behaviour space. Safety merely asserts

that the input list becomes smaller through time and the output list gets bigger. On its own the safety condition does not describe what items get added to the output list — this is implicit in the liveness condition. If the output list must eventually have a value that is the same as the initial value of the input list *and* it never gets any smaller then all the items added to the output list must have originally been in the input list and they must be added to the output list in the same order as they were removed from the input list. If a rogue item was added to the output list that was never in the input list then in order to fulfill the liveness condition that item must be first removed from the output list, hence contravening safety.

There is therefore some idea of safety implicit in the liveness condition. This however is not a problem as such. The important point is that overall the statement of requirements is clear to the reader. It is worth repeating that clarity to the reader is more important than rigorously slotting the requirements into safety and liveness boxes.

5.4.2 Are the requirements abstract enough?

We have described in an abstract way the behaviour of a system that processes items in an ordered manner. We can now step back from this description and see if there is some even more abstract system of which this is a specialised instance.

The use of sequences to model the system ensures that ordering is maintained in the system, but we suggest in certain contexts this ordering may be restrictive. Consider a multi-processor system; items are taken from the input and processed. If we have more than one processor processing items then there is no guarantee that the first item removed will be the first to be fully processed; such things depend on the speed of and resources available to each processor. An item cannot be passed into the output list until it has been processed *and* all the items taken from the input list before it have been processed. The throughput speed is then determined by the speed of the slowest processor and this may be undesirable. A more general system is one in which items are passed from the in-process list to the output list (possibly) as soon as they have been processed. In such cases, however, we cannot guarantee that the ordering of the output list will be the same as the input list. This ordering may be crucial or it may not be, this is a system design decision, but the model we have described above is not abstract enough to allow this design decision; it has some (but not much) implementation bias. The answer to

the question 'are the requirements abstract enough?' is of course context dependent, but it is always a good idea to start with requirements that are as abstract as possible.

Therefore we redescribe the requirements using multisets or bags so that we can ignore the idea of ordering. A bag is a set that allows multiple members, or, more pertinently in this situation, is a sequence without the ordering.

5.4.3 Redescribing the requirements using bags

The redescription of the system using bags is a fairly straight forward process; the ideas we formalised using sequences still apply, we are just re-expressing them in a more abstract manner. *in* and *out* are bags of *Items* rather than sequences.

$$in{:}\mathcal{B}(Item) \tag{5.8}$$
$$out{:}\mathcal{B}(Item) \tag{5.9}$$

(where $\mathcal{B}(X)$ denotes all the sub-bags of the elements in X in a similar way to how $\mathcal{P}(X)$ denotes all the sub-sets of X. See B.3.)

Redescribing the liveness condition is simply a case of saying that *out* is initially an empty bag and *in* is eventually an empty bag. Otherwise the condition is unchanged.

$$live \mathrel{\hat{=}} \exists x \bullet \begin{pmatrix} in = x \\ \wedge \\ out = \emptyset \end{pmatrix} \wedge \Diamond \begin{pmatrix} in = \emptyset \\ \wedge \\ out = x \end{pmatrix} \tag{5.10}$$

($\emptyset$ denotes an empty bag.)

The safety condition is simpler than its sequence counterpart because ordering is no longer important, hence we do not need to worry about prefixes and suffixes. We only need to say that *in* and *out* get smaller and larger respectively.

$$\begin{aligned} safe \mathrel{\hat{=}} \Box(\exists a, b \bullet \\ in = a \wedge \Box(in \subseteq a) \wedge \\ out = b \wedge \Box(b \subseteq out)) \end{aligned} \tag{5.11}$$

The relation $\subseteq$ denotes 'sub-bag or equal' in a similar way to sub-set or equal.

The more abstract requirements are the conjunction of the safety and liveness conditions.

$$reqs \mathrel{\hat{=}} live \wedge safe \tag{5.12}$$

5.4.4 Requirements engineering

We have shown the design process moving in the opposite direction (from less to more abstract) to that normally described in the literature. This shows another important use for formal methods; not only are they useful in moving toward more concrete descriptions, their abstract descriptions allow us to analyse the problem space in a more detached manner. In this case we are only worrying about how items move through a system and we have ignored all other issues. In doing so it became clear that moving items in an ordered manner was a special case of moving items in no particular order and so we reformalised the problem space in this more abstract way. If we were working with a more concrete model then such an insight may have been obscured by detail.

In 'real life' design situations it will not just be a simple case of moving in a linear fashion down the design process, but there will be cycling in the process. Expense is incurred when large cycles are needed, small cycles (such as the one we have just described) are fairly painless and cheap. A formal approach will hopefully reduce the amount and size of the cycling involved in the design process (but it will not eliminate it).

5.5 Specifying systems

Now we can start to consider a system that is going to fulfill the requirements we have described. The specification is going to be of the Pnueli style we previously discussed. We need to describe the legal initial states for the system and pieces of functionality that move items about the system in a legal manner. We can make design decisions about the system and then assure ourselves that those decisions are consistent with the requirements. This would preferably be done incrementally by constructive techniques or, less preferably, by using retrospective verification (but this is more difficult).

First of all we make the decision that there is a 'pool' of items which are the items that are currently being processed by the system.

$$pool{:}\mathcal{B}(Item) \tag{5.13}$$

Items are read from the input list and passed into the pool, once processed, items are taken from the pool and written to the output list.

The initial property of the system describes the pool and output list as being empty.

$$\begin{aligned} init \triangleq\ & out = \emptyset \wedge \\ & pool = \emptyset \end{aligned} \tag{5.14}$$

Now we make the decision that there are two pieces of functionality, one that moves items from *in* to *pool* called *take* and one called *put* that moves items from *pool* to *out*.

To guarantee liveness we need to specify that *put* and *take* occur whenever they can; once enabled they should occur, in other words they are obligatory. We denote that functionality is obligatory by enclosing it in square brackets. Hence the system specification looks like...

$$sys \triangleq init \wedge \Box[put \vee take] \tag{5.15}$$

This is the system design, but so far we have not discussed the effect of the functionality. In TLA or similar formalisms the functionality would be described as actions, or relationships between states. For reasons we shall go into in depth in the next chapter we do not take actions to be the unit of functionality in RSSL, but we use a more operational unit called a 'computation'. We digress here to introduce computations, but a fuller treatment is given in the next chapter.

5.5.1 An introduction to computations

A computation is a unit of functionality that reads data from the 'public' (or shared) state space to some 'private' space (which is some space not shared by any other computations). Processing

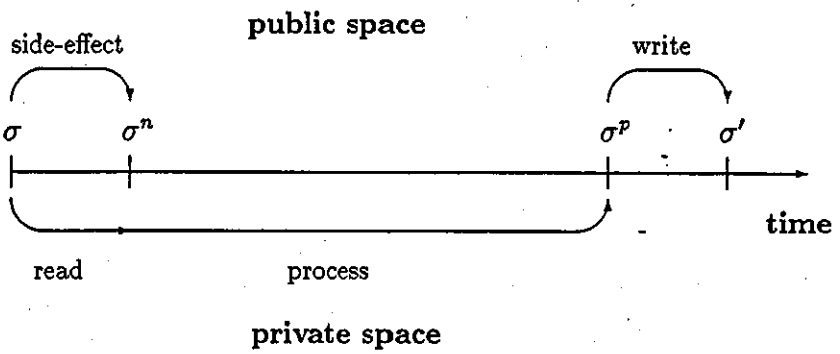


Figure 5-4: A computation being performed

then takes place on this private space and then once the processing is completed the shared state space is updated according to the result of the processing. A computation therefore passes through three phases;

- read,
- process, and
- write.

We also allow for 'destructive' reading of the public space where the value of the public space is altered by the act of reading it. We call this the 'side-effect'.

Consider figure 5-4. It shows how a computation is performed; each phase is bounded by two states, known as...

- the 'start state', denoted by an undecorated sigma σ in the figure,
- the 'next state', denoted by σ^n in the figure which is the value of the state after the state in the start state is copied to the private space and the side-effect has occurred,
- the 'penultimate state', denoted by σ^p in the figure which is the value of the state some finite time after the next state and after the process phase has completed (the process phase may be 'null'; it may have no effect on the private space and therefore take no time), and

- the 'final state', denoted by σ' in the figure, which is the value of the state after the public space has been updated according to the value of the private space.

The horizontal line represents the passage of time. Above the line is the state space that is public and shared by other computations, below the line is the space that is private to the computation.

In order to prevent interference between concurrent computations the step from the start state to the next state and the step from the penultimate state to the final state are atomic; in other words only one read or write phase can occur at a time, but several process phases can occur concurrently.

So let us now express *take* and *put* as computations. We can express computations using five clauses (in the next chapter we show how we can express complicated computations more clearly using eight clauses). Firstly we list all the variables that make up the public space. The public space is divided into the input space and the output space. The variables listed in the input space can only have their value altered in the read phase by the side-effect, likewise variables in the output space can only have their value changed in the write phase. In the case of *take* the input space is the variable *in* and the output space is *pool*.

Next we describe the enabling condition, a property that describes under what circumstances the computation can take place. In the case of *take* we can only remove items from *in* iff there are items in it *i.e. in* is not an empty bag...

$$in \neq \emptyset$$

Next we describe the side-effect (if any); the relationship between the input space in the start and next state. We denote values of variables in the start state by undecorated variables and in the next state by variables decorated by ^{n}. *take*'s side effect is to remove one item from *in*, therefore formally...

$$\exists i : Item \bullet in = in^n \uplus \{i\}$$

In words; 'there is an item i which is in the input bag in the start state, but is removed from the input bag in the next state.' The operator $\uplus$ is the bag union operator.

Finally we describe the 'outcome' relation which essentially describes the read, process and write phases together. It is a relationship between the start, penultimate and final state. *take*

has a null process phase, it simply adds the item removed from *in* by the side-effect into *pool*. If i is the item removed by the side-effect then the outcome relation is...

$$pool' = pool^p \uplus \{|i|\}$$

We bring these four clauses together using a VDM-like notation.

$$\begin{aligned} take \triangleq\ & \textbf{input} : in \\ & \textbf{output} : pool \\ & \textbf{enabled\_by} : in \neq \emptyset \\ & \textbf{side-effect} : \exists i : Item \bullet in = in^n \uplus \{|i|\} \\ & \textbf{outcome} : pool' = pool^p \uplus \{|i|\} \end{aligned} \tag{5.16}$$

Note that the scope of the existentially quantified variable i carries over all the remaining clauses. Generally this is always the case unless we explicitly delimit the scope to particular clauses using bracketing. If we were to quantify a variable in the enabling condition then its scope would carry to the side-effect and outcome clause.

Having defined *take* we can now define *put* in a very similar way.

$$\begin{aligned} put \triangleq\ & \textbf{input} : pool \\ & \textbf{output} : out \\ & \textbf{enabled\_by} : pool \neq \emptyset \\ & \textbf{side-effect} : \exists i : Item \bullet pool = pool^n \uplus \{|i|\} \\ & \textbf{outcome} : out' = out^p \uplus \{|i|\} \end{aligned} \tag{5.17}$$

In words; '*put* is enabled by there being items in *pool*. Its side effect is to remove one of those items and its outcome is to put that item into *out*.'

This completes the specification for the system.

5.5.2 Does this specification fulfill the requirements?

Having proposed this specification we need to show that it is indeed consistent with the requirements. In a fully rigorous development process we would formally prove the consistency and as we have expressed both the requirements and the specification in temporal logic we are confident that we can express this proof using temporal logic methodologies. However an involved proof

is not our concern here so we discuss rather than prove the consistency.

It is crucial to note that the specification not only describes what the system can do, it also expresses *everything* the system can do. Behaviour that is inconsistent with the specification will not be allowed in the system.

Hence we can satisfy ourselves of the safety condition because the changes to the system are caused by *put* or *take* and these computations only move items around, they do not create or destroy them.

put and *take* describe how one item moves through the system. We have made the design decision that only one item at a time moves from one space to the next, hence we have narrowed the set of possibilities in the way we described in section 5.2. We could have defined more general *puts* and *takes* which move several items at a time, but this would not have illustrated a strict refinement.

Lastly the obligatory nature of the computation assures that the final state as defined in the liveness condition will eventually be reached as items will be removed from *in* whenever there are items to be removed and will eventually all finish up in *out*.

5.5.3 A quick review of how far we have got

So far we have described the requirements for a system, then reviewed those requirements and widened them into requirements for a more general class of systems.

We then proceeded to specify an abstract system which fulfilled those requirements and discussed how that abstract system fulfills those requirements.

However the requirements and specification only describe how items move through the system, it says nothing about the actual effect of processing the those items. It only describes the kernel functionality which moves items through the system; we could add a layer of environment which places items in *in* to be processed and removes items from *out* once they have been processed. Furthermore it does not describe how quickly we want the computations to occur or how many processors we expect the system to use.

We shall look at these issues in the next section.

5.6 Developing the system specification

5.6.1 The effect of processing

So far we have only described how items move through the system, now we can look at how they are actually processed. Assume that processing one item is described by the function $f : Item \rightarrow Item$. The aim of the system is to apply f to each item initially in *in*. We can say that none of the items in *in* have had f applied to them and all the items in *out* have.

In order to describe the effect of processing we introduce a function *fbag* on bags that returns the bag of items that have had f applied to them.

$$\begin{array}{l} fbag : \mathcal{B}(Item) \rightarrow \mathcal{B}(Item) \\ fbag(items) \triangleq \{\!| f(i) \mid i \in items |\!\} \end{array} \tag{5.18}$$

Restating the requirements

Using this relationship we can restate the requirements to ensure that f has been applied to every item in *out*. Liveness states that eventually *out* contains all the items in the initial value of *in* such that f has been applied to each item.

$$live \triangleq \exists x \bullet \begin{pmatrix} in = x \\ \wedge \\ out = \emptyset \end{pmatrix} \wedge \Diamond \begin{pmatrix} in = \emptyset \\ \wedge \\ out = fbag(x) \end{pmatrix} \tag{5.19}$$

Safety which only describes how the input and output bags decrease and increase through time does not need to be restated to take into account the processing of items.

We can now look at two ways of specifying a system that are consistent with these new requirements. Both are 'correct', but the second is a little more concrete and looks more like an actual system. It is worth noting that though both are consistent with the requirements they are not consistent with each other, (neither is a correct refinement of the other) they could be thought of as representing the start of two separate design processes resulting in different designs.

Restating the specification (first pass)

The first specification simply reworks the *put* computation so that the processed version of an item is put into *out*.

$$
\begin{aligned}
put \triangleq\ & \textbf{input} : pool \\
& \textbf{output} : out \\
& \textbf{enabled\_by} : pool \neq \emptyset \\
& \textbf{side-effect} : \exists i : Item \bullet pool^{n} = pool \setminus \{\!|i|\!\} \\
& \textbf{outcome} : out' = out^{p} \uplus \{\!|f(i)|\!\}
\end{aligned}
\tag{5.20}
$$

This is straight forward and obviously consistent with the requirements; but is it reasonable as a model of a system? We are effectively saying that processing occurs as part of the *put* computation. It would make more sense to split processing and putting, and this is what we do in the next formulation of the system.

Restating the specification (second pass)

Instead of simply taking items from *in* this new system specification takes items and changes them into 'processing items'. A processing item is an item along with a Boolean flag which denotes whether it has been processed yet. These processing items are passed into the pool by *take*, once in the pool each processing item has its item converted to the processed form and its flag set to true. The *put* computation selects processed items from the pool, strips away the flag and places the item in *out*.

So, *pool* is redefined as a bag of processing items...

$$pItem \triangleq Item \times \mathbb{B} \tag{5.21}$$

$$pool : \mathcal{B}(pItem) \tag{5.22}$$

The specification states that the computations that are obligatory are *take*, *convert* and *put*...

$$sys \triangleq init \wedge \Box[take \vee convert \vee put] \tag{5.23}$$

(*init*, that says that the pool and output bag are initially empty does not need changing)

take takes items from *in* and places them as processing items in *pool*

$$\begin{aligned} take \triangleq\ & \textbf{input} : in \\ & \textbf{output} : pool \\ & \textbf{enabled\_by} : in \neq \emptyset \\ & \textbf{side-effect} : \exists i : Item \bullet in = \{\!|i|\!\} \uplus in^n \\ & \textbf{outcome} : pool' = pool^p \uplus \{\!|(i, \mathsf{false})|\!\} \end{aligned} \tag{5.24}$$

...*convert* processes items in the pool...

$$\begin{aligned} convert \triangleq\ & \textbf{input} : pool \\ & \textbf{output} : pool \\ & \textbf{enabled\_by} : \exists i : Item \bullet (i, \mathsf{false}) \in pool \\ & \textbf{side-effect} : pool = pool^n \uplus \{\!|(i, \mathsf{false})|\!\} \\ & \textbf{outcome} : pool' = pool^p \uplus \{\!|(f(i), \mathsf{true})|\!\} \end{aligned} \tag{5.25}$$

...and *put* places converted items in *out*...

$$\begin{aligned} put \triangleq\ & \textbf{input} : pool \\ & \textbf{output} : out \\ & \textbf{enabled\_by} : \exists i : Item \bullet (i, \mathsf{true}) \in pool \\ & \textbf{side-effect} : pool = pool^n \uplus \{\!|(i, \mathsf{true})|\!\} \\ & \textbf{outcome} : out' = out^p \uplus \{\!|i|\!\} \end{aligned} \tag{5.26}$$

5.6.2 Number of processors and timing

So far we have made little mention of the number of processors needed to implement the system. This is deliberate; we assert that our specification technique is neither sequential or concurrent, it is abstract enough that the decision about the number processors available is a design decision. The semantics for the language describe the behaviour of a system consistent with the specification in a way that is abstract from the machine(s) on which the implementation is to be run.

Furthermore we have said nothing about how long it takes for items to pass through the

system; liveness assures that items move through the system in a finite time, but other than that we have said nothing about time. Typically, specification techniques abstract away from time because processing time is usually very machine dependent. However reactive systems (especially safety critical systems) may need to make reference to time. For example a reactive system that responds to a gas burner, turning the gas supply off when the flame goes out will need to respond in an explicit time in order to prevent a dangerous build up of gas [99]. Furthermore usability in interactive systems can rely heavily on the response speed of the kernel. We therefore include apparatus for explicitly discussing time.

Number of processes

Let us assume that it takes one process to perform one computation and we can have have N processes at a time. We can maintain a count p of the number of processes currently working and include in the enabling condition for each computation a condition stating that a computation can only be launched when $p < N$. Furthermore we conjoin the fact that p is zero when the system starts.

Formally...

$$N, p{:}\mathbb{N} \tag{5.27}$$

$$init \mathrel{\hat{=}} \ldots \wedge p = 0 \tag{5.28}$$

$$\begin{aligned} take \mathrel{\hat{=}} & \textbf{input} : \ldots, p \\ & \textbf{output} : \ldots, p \\ & \textbf{enabled\_by} : \ldots \wedge p < N \\ & \textbf{side-effect} : \ldots \wedge p^n = p + 1 \\ & \textbf{outcome} : \ldots \wedge p' = p^p - 1 \end{aligned} \tag{5.29}$$

...and so on for *convert* and *put*.

The side effect increments p, effectively claiming a process for the computation and does not release it until its write phase is completed. This a simple model of process allocation; we

could provide a more complicated model that allows more than one process to work on a single computation and so on.

Timing considerations

To describe timing considerations we use the special variable t. t is a real number that behaves in certain ways, most notably it always increases. This makes t distinct from all the other variables in a specification which only change their value when its is explicitly specified that they can. Time changes its value implicitly.

Assuming that the time units are seconds we can specify that *take* occurs in at most 5 seconds as follows...

$$\begin{aligned} \mathit{take} \triangleq\ & \textbf{input} : \ldots \\ & \textbf{output} : \ldots \\ & \textbf{enabled\_by} : \ldots \\ & \textbf{side-effect} : \ldots \\ & \textbf{outcome} : \ldots \wedge t' \leq t + 5 \end{aligned} \tag{5.30}$$

Obviously we should not do impossible things with time; such as $t' < t$.

Timing considerations should only be brought into an abstract specification if really necessary. We must be aware that including timing constraints may limit us to systems that are very difficult to implement, timings on computations may simply to be too fast for a reasonable machine to perform. Say we place a time constraint on *convert*, then we need to be sure that the function f can be performed in that time. If f is simple then this poses no problem, but what if *Items* are data lists and f is a sort? The time to perform f is exponentially related to the size of each item and putting a simple time limit on it can be very restrictive.

5.6.3 Including the environment

So far we have looked at a fairly static system that simply processes a given number of items and then stops. Now we widen the boundaries of our system and assume we have another agent that places items in *in* and removes them from *out*. What we have described so far is the system kernel and the agent that does the putting and removing is the system environment.

Redefining the requirements

Safety and liveness are now two sides of the same coin; safety states that every item that gets into the output bag must originally have been in the input bag and liveness states the reverse; every item in the input bag must eventually get into the output bag. So we can express the two in one formula (hence demonstrating our point about safety and liveness becoming easily intertwined)...

$$safelive \hat{=} \biguplus_{\Diamond\, in=x} x = \biguplus_{\Diamond\, out=y} y \tag{5.31}$$

In words 'the union of all the items that are ever in *in* is the same as the union of all items ever in *out*'.

The initial condition was implicit in the old liveness condition, this is not the case now, so we need to express it explicitly...

$$initial \hat{=} out = \emptyset \tag{5.32}$$

So our new requirements are...

$$req \hat{=} safelive \wedge initial \tag{5.33}$$

Redefining the specification

put, *convert* and *take* do not need to be redefined, but we need to be aware that square brackets around a computation not only denote obligation on the computations but also 'fairness'. When we had a finite number of items in the system then we did not need to worry about fairness. We may have proposed an implementation strategy that takes *all* the items out of the input bag first, then processes them, then puts them in the output bag. This would be a successful strategy if we were sure that there would only be a finite number of items placed in the input bag. If an infinite number of items were placed in the input bag we would keep indefinitely launching

takes to deal with them, such that *convert* and *put* never get the chance to launch; that would be an unfair system. All obligatory computations are assumed to be fair, *i.e.* overall they must roughly occur as often as they are enabled. Fairness is defined as being 'if a computation is enabled infinitely often then it must be launched infinitely often.[2]'

There are several implementation strategies that guarantee fairness such as 'round-robin' polling *etc.*

Optional computations

The two computations *place* and *remove* place items in the input bag and remove them from the output bag respectively.

$$\begin{aligned} place \mathrel{\hat{=}} & \textbf{output} : in \\ & \textbf{outcome} : \exists i : Item \bullet in' = in^p \uplus \{\!|i|\!\} \end{aligned} \tag{5.34}$$

$$\begin{aligned} remove \mathrel{\hat{=}} & \textbf{output} : out \\ & \textbf{enabled\_by} : out \neq \emptyset \\ & \textbf{outcome} : \exists i : Item \bullet out^p = \{\!|i|\!\} \uplus out' \end{aligned} \tag{5.35}$$

We do not wish to assume that these computations are eagerly processed, indeed we would make very little sense if they were. In particular *place* is always enabled (if there is no explicit enabling condition then it is held to be true) and we do not wish to say that *place* is always occurring. Obviously a computation cannot occur unless it is enabled; obligatory processing means that a computation is processed whenever it is enabled (subject to processing resources being available), optional processing means that a non-deterministic choice about whether processing occurs or not is made whenever a computation is enabled. Optional processing is denoted by enclosing the computation in angle brackets, hence our specification is...

$$sys \mathrel{\hat{=}} init \wedge \Box[put \vee convert \vee take] \wedge \Box\langle place \vee remove \rangle \tag{5.36}$$

[2]This is the 'strong fairness' definition. There are other definitions that are not our concern here. See [55].

Equation 5.36 shows the canonical form for RSSL specifications; an initial predicate, a disjunction of obligatory kernel computations and a disjunction of optional environment computations.

5.6.4 A reactive system

It is easy to see that we can pair certain optional and obligatory computations together. Consider *place* and *take*; the outcome of *place* enables *take* and the obligatory nature of *take* ensures it occurs. We can therefore say that *place* 'causes' *take*.

We can capture this causality in a 'reaction'; an explicit pairing of optional and obligatory computations. The reaction formed by pairing *place* and *take* is denoted $place \fatsemi take$. As *place* is optional then the whole reaction is optional too. The following two specifications are equivalent;

$$sys \hat{=} init \wedge \Box[take] \wedge \Box\langle place \rangle \tag{5.37}$$

$$sys \hat{=} init \wedge \Box\langle place \fatsemi take \rangle \tag{5.38}$$

In the second equation the notion of obligation is subsumed into the notion of reaction. We use reactions to delimit another layer of 'typicalness' in our reactive systems; typically for each optional environment computation, called an 'invocation' there is a corresponding obligatory computation, called a 'response', that is caused by it. A specification in the 'reaction style' is simply a device for making this causality between computations clearer.

However we need to be explicit about which environment computation causes which obligatory computation. To do this, as well as an enabled by clause in the obligatory computation we add an 'invoked by' clause which explicitly states which optional computation is causing it. Hence, in the reaction style of specification, we would rewrite *take* as follows...

$$
\begin{array}{l}
take \triangleq \vdots \\
\qquad \textbf{invoked\_by} : place \\
\qquad \textbf{enabled\_by} : in \neq \emptyset \\
\qquad \vdots
\end{array}
\tag{5.39}
$$

In order to show the need for explicit invocation clauses consider the following cautionary specification.

A cautionary specification of a reactive system

A display panel in a nuclear power plant control centre shows the condition of a valve using some icon. The icon can have two states; open and closed as can the valve itself.

The controller can open or close a valve by manipulating the icon (by some means we are not too interested in) and there is a safety device in the reactor that can close or open the valve to prevent explosions, leaks or other undesirable occurrences in times of crisis.

The requirements (of which we shall give no details) say that the icon should represent the condition of the valve as often as possible. Note that the icon and valve cannot have the same value *all* the time; there must be some (minimal) time lag between one changing an the the other changing in response.

There are two kernel computations that change the value of the icon or valve so that the two match each other, and there are two environment computations one representing the controller changing the value of the icon and one representing the safety system changing the value of the valve. We assume that system starts in a state where the icon and valve have the same value. All this is shown in figure 5-5.

If we assume that the outcomes occur as fast as possible we still have a very big problem with this specification. Imagine a situation where the valve is open, there is an emergency in the plant and safety system shuts the valve. Now the icon and the valve have different values, so *updateIcon* is enabled and as it is obligatory then the icon should be updated as we would expect. However, both *updateIcon* and *updateValve* have the same enabling condition, so we could have a situation where in order to get the valve and icon to have the same value, *updateValve* is launched with possibly disastrous consequences.

$$icon{:}\{\text{Open}, \text{Closed}\} \quad (5.40)$$
$$valve{:}\{\text{Open}, \text{Closed}\}$$

$$badSpec \triangleq init \wedge \Box[updateValve \vee updateIcon] \wedge \Box\langle controller \vee safety \rangle \quad (5.41)$$

$$init \triangleq icon = valve \quad (5.42)$$

$updateValve \triangleq$
input : *icon*
output : *valve*
enabled_by : $icon \neq valve$
outcome : $valve' = icon$ (5.43)

$updateIcon \triangleq$
input : *valve*
output : *icon*
enabled_by : $icon \neq valve$
outcome : $valve = icon'$ (5.44)

$controller \triangleq$
output : *icon*
outcome : $icon' \neq icon$ (5.45)

$safety \triangleq$
output : *valve*
outcome : $valve' \neq valve$ (5.46)

Figure 5-5: An innocuous looking, but very dangerous specification

Obviously *safety* should be paired with *updateIcon* and *controller* should be paired with *updateValve*. We need to add more conditions to the enabling conditions to make clear which computation should be launched. We could do this with Boolean flags that are set to true when an invocation is launched and the enabling condition for the response includes them being true. However the use of flags like this tends to clutter the formulae so we provide reactions as a syntactic sugar to hide these flags. (In fact as we shall see in the next chapter, reactions hide counters of how many invocations there have been.)

An important point in favour of the reaction style of specification is that it discourages us from defining specifications which look satisfactory, such as figure 5-5, but which in fact are not. The problem with figure 5-5 is not immediately obvious and one of the main 'selling points' of formal approaches is that they lay problems open to inspection because of their abstract nature.

It would be (more) difficult to make this mistake when describing the system in the reaction style. Again we return to one of our running themes; techniques for describing typical systems whilst still retaining enough generality to describe atypical ones too and making it easier to describe typical systems so as to expose atypicalities.

An improved specification is shown in figure 5-6. The *controller* and *updateValve* computa-

$$goodSpec \triangleq init \wedge \Box(controller \fatsemi updateValve \vee safety \fatsemi updateIcon) \quad (5.47)$$

$$init \triangleq icon = valve \quad (5.48)$$

updateValve $\triangleq$
input : *icon*
output : *valve*
invoked_by : *controller*
enabled_by : $icon \neq valve$
outcome : $valve' = icon$ (5.49)

updateIcon $\triangleq$
input : *valve*
output : *icon*
invoked_by : *safety*
enabled_by : $icon \neq valve$
outcome : $valve = icon'$ (5.50)

controller $\triangleq$
output : *icon*
outcome : $icon' \neq icon$ (5.51)

safety $\triangleq$
output : *valve*
outcome : $valve' \neq valve$ (5.52)

Figure 5-6: An improved specification

tions are explicitly paired to form a reaction, as are *safety* and *updateIcon*. Both the responses have 'invoked by' clauses added to show which optional computations invoke them.

Note that this system bears a marked resemblance to 'direct manipulation' systems where both the user and some underlying functionality have access to some representation that they can both manipulate. We have shown a simple specification for maintaining consistency between the representation and the functionality. See for example, the similarity between the specification here and the specification of a scroll bar presented in [23].

A specification in the reaction style is a collection of optional reactions. The notion of obligation is 'hidden' in the reactions.

5.7 Some implementation strategies

We now look some sample implementations of the system. There are, as yet, very few tried and tested techniques[3] for program derivation in concurrent systems. Therefore we must proceed with caution. We *suggest* an implementation structure and show how we can produce some implementations based on our specifications. There is however, as yet, no generally agreed

[3]At least not 'tried and tested' to the extent that VDM's refinement calculus has been tried and tested, for example.

derivation from specification to implementation; we propose an implementation and then would need to prove that it fulfills some specification by a post hoc verification.

Such post hoc verifications can be notoriously laborious, so we suggest that the system designer refines the specification as far as possible before making the step into implementations; this should ease the discharging of proof obligations.

5.7.1 Specification refinement

We know that a specification ψ_1 is a refinement of ψ_2 if ψ_1 does not do anything considered illegal in ψ_2 and does everything that is expected of it. In other words, the refinement respects the safety and liveness conditions implicit in the specification it refines.

Ensuring safety is simply a case of showing that the behaviour described by a refinement is a sub-set of the behaviour described by its specification. Actually proving this would be very arduous, but as our specification language is within the bounds of TLA we contend that we can import the proof methodology of TLA and then we can equivalently prove that $\psi_2 \Rightarrow \psi_1$. Liveness is more involved and is described in the next chapter.

A system designer can get considerable mileage in refining a specification in this manner; he can progressively split computations into several more refined computations and he can also introduce ideas about the behaviour of internal state by existentially quantifying the internal state, so long as the refinement is correct with respect to the more abstract specification. This refinement process should keep the new, refined specifications within the bounds of our specification style; they should still describe an initial predicate and sets of obligatory and optional computations, even if the sets are very large and the computations very detailed and concrete. Complexity should only be introduced into our specifications in two ways; more computations and more complicated computations.

At some point we need to take the step from a description of a system in RSSL style to a description that looks like a programming language. First we must be sure which parts of the system we are going to implement.

5.7.2 Splitting the kernel from the environment

Recall that the environment is part of the system that already exists; we do not therefore need to worry about implementing it. The system we wish to implement is specified by...

$$sys \triangleq init \wedge \Box[take \vee convert \vee put] \wedge \Box\langle place \vee remove\rangle$$

...therefore we assume that we already have a system that behaves as...

$$environ \triangleq \Box\langle place \vee remove\rangle \tag{5.53}$$

...and we need to implement a system that behaves as...

$$kernel \triangleq init \wedge \Box[take \vee convert \vee put] \tag{5.54}$$

Actually *init* may be set up by the kernel or environment or a combination of the two.

5.7.3 An implementation structure

Our implementation structure for the kernel is based on a construct similar (but *not* the same) as the **do** loop suggested by Dijkstra [36].

$$\begin{array}{l} init_3^? \\ \textbf{do} \\ \quad in \neq \emptyset \rightarrow T \\ \quad [] \\ \quad \exists i : item \bullet (i, \mathsf{false}) \in pool \rightarrow C \\ \quad [] \\ \quad \exists i : item \bullet (i, \mathsf{true}) \in pool \rightarrow P \\ \textbf{else} \\ \quad \mathsf{true} \rightarrow \textbf{skip} \\ \textbf{od} \end{array} \tag{5.55}$$

...where T, C and P are code fragments representing the side-effects and outcomes of *take*, *convert* and *put* respectively.

Once we have established the initial conditions the program enters an infinite loop where

the guards are evaluated (the guards being synonymous with the enabling conditions) and a single computation is selected from those that have their guards evaluate to true. If no guards evaluate to true the **else** clause is performed; nothing happens and the system waits for the environment to trigger further computations. We assume that computations are not simply non-deterministically selected once their guards are true but that the selection is guided by the fairness conditions. We assume a 'fair scheduler' on the **do** loop. Such things are described in [55]. Note that this program is non-terminating. We consider termination to effectively be the indefinite selection of the **else** clause.

We can implement *init* by;

$$(out, pool) := (\emptyset, \emptyset)$$

For each of the code fragments we need some code that has the effect of the side-effects occurring 'next'; control does not pass back to the **do** loop until the side-effect is completed and a train of events is put in place that ensures the eventual completion of the outcome.

5.7.4 Implementation using sequential constructs

We can place the side-effect and outcome in sequence as long as the side-effect is not destructive. In the case of *take* the side-effect is destructive so we need to be careful with the implementation. We can copy the relevant parts of the global space to some internal variables, perform the side-effect and then perform the outcome based on the value of the internal variables. So T may have the form...

$$\begin{array}{l} i := \mathbf{removefrom}(in)^{\S}_{\S} \\ in := \mathbf{remaining}(in)^{\S}_{\S} \\ pool := pool \uplus \{\!|(i, \mathsf{false})|\!\} \end{array} \tag{5.56}$$

The operation **removefrom** is deterministic — it removes an item from a bag, but given the same bag, it always removes the same item from the bag. It is paired with the operation **remaining** which returns the bag with the item selected by **removefrom** removed.

Note that control does not return to the **do** loop until both the side-effect and outcome have

been completed, hence only one instance of *take* can be performed at a time using this sort of implementation.

A generic code fragment for implementing computations in a sequential programming language is;

copy global variables to internal ⨟ side-effect ⨟ outcome using internal variables

We would, of course, implement the other two fragments C and P in this way too.

5.7.5 Implementation using concurrent constructs

Assume now that we have a language that supports the following constructs...

- $P \parallel S$ — parallel composition. P and S are started in the same instant and run in parallel. The construct does not complete until both P and S are completed.
- **fork**(P) — a process is forked to deal with P. P runs concurrently to whatever else is going on in the system.

...we can implement T as follows...

$$\begin{array}{l} i := \textbf{removefrom}(in) \fatsemi (in := \textbf{remaining}(in) \parallel \\ \textbf{fork}(pool := pool \uplus \{\!|(i, \mathsf{false})|\!\})) \end{array} \tag{5.57}$$

Global variables are copied to internal space and then the side-effect and outcome are performed in parallel. The side-effect is performed immediately and a process is forked to deal with the outcome. Because the time taken to perform a **fork** is minimal control returns to the **do** loop as soon as the side-effect is completed.

A generic form code fragment implementing computations is;

copy to internal ⨟ (side-effect ∥ **fork**(outcome using internal))

5.8 Summary

We have taken care to describe our notion of reactive systems and have suggested how reactive systems relate to interactive systems.

A reactive system is a collection of interacting sub-systems. The sub-systems can be divided into environment sub-systems and kernel sub-systems. The environment sub-systems are highly non-deterministic and typically are the parts of the system that are already extant. Typically, the kernel sub-systems are automated and fairly deterministic in their behaviour. It is usually the kernel that is designed and built.

To conclude this chapter let us run through a generalised system and see how parts of it fit into a design process.

5.8.1 A design process stated generally

Requirements

Requirements describe the problem space that we need a system to fulfill. Typically requirements are expressed as a conjunction of safety and liveness conditions.

$$req \triangleq safe \wedge live$$

A safety condition states that bad things never happen and a liveness condition states that something good does eventually happen. Reactive systems may require a slightly more complex expression of liveness condition. Liveness in the kernel is dependent on the environment — something good only need eventually happen if the environment requests it. This unfortunately blurs the distinction between requirements and specifications. Ideally requirements should be stated in complete isolation to any ideas of what system fulfils them. However liveness stated in this way needs to have *some* idea of system in order to make the distinction between kernel and environment.

The requirements are generated by requirements engineers.

System specification

Specifications describe systems which fulfill requirements. A system specification is a conjunction of an environment specification and a kernel specification, sometimes known simply as the specification.

Our model of reactive system proposes that the environment makes requests of the kernel by enabling the requested functionality. Once enabled the kernel functionality must respond by performing that enabled functionality (assuming there are processing resources available).

The system specification lists all the pieces of functionality that the environment can perform and all the pieces of functionality that the kernel must perform when enabled. Also included in the system specification is a description of the legal initial states. If $\{e_1, \ldots, e_n\}$ is all the environment computations, $\{k_1, \ldots, k_m\}$ is all the kernel computations and *init* is the description of legal initial states then the system specification will have the form...

$$sys \triangleq init \wedge \Box[k_1 \vee \ldots \vee k_m] \wedge \Box\langle e_1 \vee \ldots \vee e_n \rangle$$

The angle brackets denote the optional nature of the environment functionality and the square brackets denote the obligatory nature of the kernel functionality. This specification will be produced by a system designer.

Functionality is modelled by a novel entity which we call a computation. A computation shows the relationship between some internal state space and the public state space. The performance of a computation passes through three stages; the read phase where data is copied from the public space to the internal and the public space may also be updated, followed by a process phase where processing is performed on the internal space alone and finally the public space is updated with the result of the process phase.

Typically environment and kernel computations can be paired together; the purpose of environment computations is to enable kernel computations. Because kernel computations are obligated to occur once enabled we can say that certain environment computations cause certain kernel computations. To make it clearer what is going on we can explicitly pair these computations together to form a reaction. If computation e causes computation k then the reaction formed by pairing the two is denoted by $e \S k$. We can then specify a system as a disjunction

of optional reactions. We call this a specification in the reaction style.

$$sys \hat{=} init \wedge \Box\langle r_1 \vee \ldots \vee r_n \rangle$$

...where for all i between 1 and n, $r_i = e_i \S k_i$. The following specification is equivalent to the one above...

$$sys \hat{=} init \wedge \Box[k_1 \vee \ldots \vee k_n] \wedge \Box\langle e_1 \vee \ldots \vee e_n \rangle$$

The reaction style specification is simply a tool for making the interplay between the environment and kernel clearer.

Refinement of the kernel

The system specification is divided into the assumptions (which capture the part of the system that does not need to be built) and the kernel specification which does need to be built. Typically the assumptions will be the part of the system captured in the optional computations...

$$assume \hat{=} \Box\langle e_1 \vee \ldots \vee e_n \rangle$$

...and the kernel specification will be the initial state description and the kernel computations...

$$kernel \hat{=} init \wedge \Box[k_1 \vee \ldots \vee k_n]$$

If the kernel is purely automated functionality then the above is the specification for the software system.

Refinement is the process of consistently adding extra description and complexity to specifications to make them more 'executable'. The extra complexity added is determined by design decisions made by the refiner. The only way of adding complexity is more complicated computations or more computations, therefore the actual form of the specification does not alter through the majority of the refinement process; only the computations in the specification alter.

A refinement process should maintain safety, (*i.e.* not introduce behaviour that is unsafe)

and ensure liveness. Safety can be thought of as the maximum a system should do and liveness can be thought of as the minimum a system should do. How we define the safety and liveness implicit conditions in a specification is involved and a way of doing so is suggested in the next chapter when we give a definition of the refinement operator $\triangleright$.

As the kernel computations represent pieces of software the process of refining them is the province of software engineers.

Implementation

Assuming that the software system specification has the following form...

$$kernel \hat{=} init \wedge \Box[k_1 \vee \ldots \vee k_n]$$

...then the following is a suggestion for an implementation framework for the system...

$init$;
do
  $k_1.E \rightarrow$ copy to internal ; $(k_1.S \parallel \mathbf{fork}(k_1.O))$
  []
  ⋮
  []
  $k_n.E \rightarrow$ copy to internal ; $(k_n.S \parallel \mathbf{fork}(k_n.O))$
else
  true $\rightarrow$ **skip**
od

...where $.E$ extracts the enabling condition from a computation, $.S$ extracts the side-effect and $.O$ extracts the outcome. $\parallel$ is a parallelism operator and **fork** generates a process fork that performs its argument.

The step from specification to implementation is still part of the software engineering processing, after that we are into the realm of the programmer.

Chapter 6

A formal semantics for RSSL

Having introduced RSSL in the previous chapter we now proceed to give a formal semantics for its notations.

6.1 Formal specification notations

Before going into details for our notations we first look at formal notations in general. In the context of this work a formal notation is a language for expressing models of system behaviour.

A formal notation is characterised by defining three mathematical objects and defining the relationship between them. These three objects are...

A model of system behaviour showing how a system develops or changes through time. This change can be modelled in several ways and in the literature has many different names; 'a trace' or 'a history' or a 'a computation'. We call such an object 'an activity' and model it as a mapping from time to system state. We define system 'behaviour' to be the set of all activities that a system can legally perform.

The syntax for the notation is usually expressed as a collection of rules for generating a set of well formed system descriptions.

A formal semantic function which takes a well formed system description and returns the behaviour that that description describes. If *Behaviour* is the set of all models of system

behaviour and D is the set of well formed descriptions generated by the syntax then a semantic function has the signature...

$$D \rightarrow Behaviour$$

The purpose of this chapter is to define the set *Behaviour*, specific instances of D for temporal logic and RSSL specifications and semantic functions for both. Section 6.2 defines the set *Behaviour*, section 6.4 defines the syntax for the simple temporal logic used in RSSL which is denoted F and a semantic function $f : F \rightarrow Behaviour$ which defines the behaviour described by elements of F. Likewise section 6.6 defines *RSSL* the set of all RSSL specifications and section 6.7 defines the semantic function $s : RSSL \rightarrow Behaviour$.

This discussion makes clear that the two notations we have so far proposed only differ in *style*; they both describe the same thing, only in different ways.

6.2 A model of real time system behaviour

There are several ways of describing or conceptualising system activity. Most common in the literature is an infinite sequence of states; $\sigma_0, \sigma_1, \sigma_2, \ldots$ The system is thought of moving discretely from one state to the next. It tends not to be clear how time is dealt with in such a model. Possibly each state is thought of as lasting for a fixed period of time, or maybe each state is thought of as lasting until there is a state change. Such models are usually the basis for specification techniques that abstract away from time altogether, hence the ambiguity.

We wish to be able to deal with time explicitly, so we intend to be exacting about how we deal with time. Pnueli [98] uses a real time model to describe system activity; time is modelled using a 'dense set'. (A dense set being defined as a set for which each pair of distinct elements has at least one element lying 'between' them. The reals are a classic example of a dense set.) This approach, we believe, has two major advantages;

- there is no fixed granularity in the model; refinement is simplified, and therefore
- using real time elegantly overcomes the 'stuttering problem' [74, section 5.1].

However, to counterbalance these advantages there is the philosophical problem of 'Zeno's paradox' (which is fully explained in section 6.2.5).

This section deals with four entities;

State — a 'snap-shot' of a system at a single instance. State is characterised by the assignment of values to the variables in a system.

Time — Abadi and Lamport [1] show that TLA can be made to deal with explicit timing issues by including time amongst the variables in a specification. We take the stance that there are enough differences between time and other variables to treat time as a special case. We wish to describe state changes explicitly, so that only state changes explicitly described in the specification can occur in the system. Having to explicitly describe all time changes in a system would lead to rather bizarre and cumbersome specifications. The major difference between state and time is that state changes must be described explicitly whereas times changes implicitly.

Activity — a model of how system state develops through time.

Behaviour — all the activities that a system can perform.

Because we are dealing with real time we need to discount 'Zeno's paradox' from our model of activity. The formalism for this is quite involved and so we relegate it to sub-section 6.2.5 at the end of this section. This sub-section can be omitted without losing track of the rest of this chapter.

Now let us look at these ideas formally.

6.2.1 State

A state is a mapping from variable names to values. We make no particular restriction as to what constitutes a variable name, but in our work we stay with convention, naming variables using finite sequences of characters. A value can be any denotation whatsoever.

We denote state by Σ. Formally...

$$Var \triangleq \ldots \tag{6.1}$$

$$Val \triangleq \ldots \tag{6.2}$$
$$\Sigma \triangleq Var \rightarrow Val \tag{6.3}$$

Following from Lamport [74, Note 1] we assume that *Val* is the set of all sets. Hence the set *Val* includes all the real numbers...

$$\mathbb{R} \subset Val$$

A useful relation over states which we make use of later is *canVary*. Given two states and a set of variable names, *canVary* holds if the two states are the same apart from (possibly) the values of the variables in the given set.

$$\begin{array}{l} canVary : \Sigma \times \Sigma \times \mathcal{F}(Var) \rightarrow \mathbb{B} \\ canVary(\sigma_1, \sigma_2, xs) \triangleq \forall v : Var \bullet v \notin xs \Rightarrow \sigma_1(v) = \sigma_2(v) \end{array} \tag{6.4}$$

In words; 'all variables not in the set xs must have the same value in σ_1 and σ_2.'

6.2.2 Time

We model time using the non-negative reals.

$$\mathsf{T} \triangleq [0, \infty] \tag{6.5}$$

...that is the (dense) set of all real numbers greater than or equal to zero.

6.2.3 Activities

An activity shows how the state of the system develops through time.

In most such models presented in the literature activities are modelled as a sequence of states. However, as we discussed earlier, what the index to that sequence represents is not clear. The index may represent the passage of a fixed unit of time, for example if α is a sequence of states then $\alpha(3)$ denotes the state after three units of time. Alternatively the index may represent the number of state changes, hence $\alpha(3)$ denotes the state after the third state change.

Both of these representations have problems; in the the first case time has a fixed lower limit of granularity and this may cause problems in refinement and in the second case time is abstracted away from altogether. In many situations this may be desirable, but in the sort of reactive and interactive systems we will be specifying time may be critical. We may wish to abstract away from explicit time, but it may be crucial that we do not.

We model activities as a partial function from time to state. Even though time itself is uncountable we only wish to consider activities that represent a countable number of state changes. A simple function from time to state does not preclude activities with an uncountable number of state changes. Such activities are not really what we are interested in. We call such a function a 'raw activity'.

$$RawActivity \triangleq \mathsf{T} \xrightarrow{p} \Sigma \tag{6.6}$$

An activity that undergoes uncountable state changes can be suffering from 'Zeno's paradox' where time advances, but by infintessimally small increments so there is a finite point of time that is never reached. Obviously it would be impossible to implement a system with such 'Zeno' activities, so we define *Activity* to be the set of all non-Zeno[1] activities.

$$Activity \triangleq \{\alpha : RawActivity \mid nonZeno(\alpha)\} \tag{6.7}$$

In words; 'an activity is any raw activity that does not suffer from Zeno's paradox.'

Henceforth when we refer to an activity we mean a non-Zeno activity.

Pictorial representations of activities

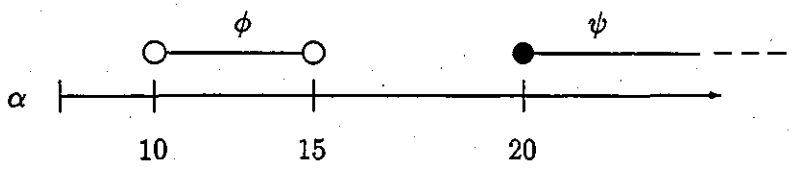


Figure 6-1: An example of an activity pictorially represented

[1]The predicate *nonZeno* is defined in section 6.2.5

It is rather more difficult to represent a real-time activity than it is represent a discrete sequence of states. Therefore we introduce a pictorial representation of activities, an example of which is shown in figure 6-1.

An activity is represented by a horizontal bar labeled by the name of the activity at the far left. Distance along the horizontal represents time. Time zero is usually included at the far left of the bar and the far right of the bar has a pointer attached to represent that the activity extends to infinite time. We can describe what is true at given times in an activity by marking the bar with vertical lines, labeling those lines underneath the bar with times and above with descriptions of what is true at this given time. We can show that descriptions hold over continuous times by marking the start time and end time and spanning the two marked times with a line labeled with the description. If the condition holds indefinitely then we mark the start time at which the condition becomes true and span the rest of the activity with a line that is dashed at the right extreme. Furthermore, if the horizontal line spanning contiguous time is terminated by an empty circle shows that the description holds immediately before or after that time, but not at that time. If the circle is solid then the description holds at precisely that time.

Figure 6-1 shows that in activity α, ϕ is true between (but not including) times 10 and 15 and that ψ is true at time 20 and holds indefinitely thereafter.

This pictorial representation is not intended to be a particularly rigorous notation, merely an aid to comprehension which is particularly useful in section 6.4 where we introduce several variations of temporal logic operators.

Activity application

As an activity is a partial function from time to state, we may have times for which no state is defined. In these cases we infer that the state is the same as it was at the most recently defined time. For example, assume we have an activity for which the state is only defined at integer times (0,1,2,3,...). The state at time 2.5 is assumed to be the same as the most recently defined time, which in this case is 2.

For this assumption to hold we need to assert that an activity has at least the state at time zero defined;

$$\forall \alpha : Activity \bullet 0 \in \mathbf{dom}\ \alpha \tag{6.8}$$

In words; 'it is always the case that an activity contains a state defined at time zero.'

We denote activity application (the state at a given time) by bold brackets — $\alpha(t)$ where $\alpha : Activity$ and $t : \mathsf{T}$. Formally...

$$\begin{array}{l} \alpha(t) \triangleq \alpha(recent) \\ \textbf{where}\ \ recent = max(\{early : \mathsf{T} \mid early \in \mathbf{dom}\ \alpha \wedge early \leq t\}) \end{array} \tag{6.9}$$

In words; '$\alpha(t)$ denotes the state in α at time *recent* where recent is the most recent time for which there is a defined state in α.'

Also it is useful to consider the state and time together as a pair, so the notation $\alpha((t))$ does this.

$$\alpha((t)) \triangleq (\alpha(t), t) \tag{6.10}$$

In words; '$\alpha((t))$ denotes the pair of state and time such that the state is the state at time t in α and the time is t.'

Variant activities

A useful equivalence on activities is *variant* which holds iff two activities may only differ by the value of a given variable.

$$\begin{array}{l} variant : Activity \times Activity \times Var \rightarrow \mathbb{B} \\ variant(\alpha, \alpha', x) \triangleq \forall t : \mathsf{T} \bullet \forall v : Var \bullet x \neq v \Rightarrow \alpha(t)(v) = \alpha'(t)(v) \end{array} \tag{6.11}$$

In words; 'at all times all the variables except x have the same value in both α and α'.'

6.2.4 Behaviour

Systems will be able to engage in many different activities. The behaviour of a system is the set of all activities that the system can engage in.

$$Behaviour \triangleq \mathcal{P}(Activity) \tag{6.12}$$

6.2.5 Non-zeno activities

An activity that is non-zeno only allows a finite number of state changes in a finite time. Another way of looking at this is to 'chop' an activity up into sections of equal time and to insist that *at most* one state change occurs in each section. As each section is of equal length, a finite piece of activity can only be split into a finite number of sections. Furthermore as at most one state change can occur in a section then there must only be a finite number of state changes in that finite piece of activity.

However, what if two variables change value at the same time; is this one or two state changes? Having two variables changing value at the same time is fine, but we do not want either of them to change value again during that portion. More subtly, we want to allow distinct variables to change value slightly out of step with one another (as most variables will do at a fine enough level of time granularity) without this 'out-of-stepness' being construed as two state changes in an infinitesimally short time.

Hence our definition of non-Zenoness is 'an activity can be divided into sections of equal length and a variable can only change its value *at most* once in each section.'

Assume that there is a predicate *atMostOne* which takes a raw activity and two times and holds true if every variable changes value at most once in that activity between the two times. We can define the predicate *nonZeno* as follows...

$$\begin{aligned} &nonZeno : RawActivity \rightarrow \mathbb{B} \\ &nonZeno(\alpha) \mathrel{\hat{=}} \exists \delta : \mathsf{T} \bullet \delta > 0 \wedge \\ &\qquad \forall n : \mathsf{N} \bullet atMostOne(\alpha, n \times \delta, (n+1) \times \delta) \end{aligned} \tag{6.13}$$

In words; 'an activity is non-Zeno if it can be divided into sections of length δ and each variable changes its value at most once in each section.'.

A variable changes its value at most once in a section if the section can be divided in two such that the value of the variable does not change in the first 'half' or second 'half' (but the variable may have different values in each half. The predicate *atMostOne* is defined based on this idea...

$$
\begin{aligned}
&atMostOne : RawActivity \times \mathsf{T} \times \mathsf{T} \rightarrow \mathbb{B} \\
&atMostOne(\alpha, t, t') \triangleq \forall x : Var \bullet \\
&\qquad \exists mid : \mathsf{T} \bullet t \leq mid < t' \wedge \\
&\qquad\quad constant(\alpha, x, t, mid) \wedge \\
&\qquad\quad constant(\alpha, x, mid, t')
\end{aligned}
\tag{6.14}
$$

In words; 'for each variable x the section between t and t' is divided into two 'halves' bounded by t and mid and mid and t' and x does not change value in both 'halves'.'

The predicate *constant* holds true if the given variable does not change value between the two given times.

$$
\begin{aligned}
&constant : RawActivity \times Var \times \mathsf{T} \times \mathsf{T} \rightarrow \mathbb{B} \\
&constant(\alpha, x, t, t') \triangleq \forall mid : \mathsf{T} \bullet t \leq mid < t' \Rightarrow \\
&\qquad\qquad \alpha(t)(x) = \alpha(mid)(x)
\end{aligned}
\tag{6.15}
$$

In words; 'the value of x is the same from time t to (and not including) t' is the same as it was at time t'.

6.3 State relationships and properties

Fundamental to our notations is the ability to describe what is or is not true about states and times. We could describe a state by listing all the variables and their corresponding values. This, of course, is cumbersome and does not allow us any degree of approximation. We need descriptions of states and times that are compact and allow approximation. Such descriptions are 'state relationships' (or relationships for short) and we make great use of them in the rest of this chapter, so we devote a section now to describing them and giving their semantics.

State relationships describe relations that either hold true or false between states. For example the TLA action $x' = x + 1$ is a state relation and the outcome clause in a computation is also a state relation. We use decorations on variables to denote the value of variables in different states.

A special case of state relationships are 'properties' which describe what is or is not true in a single state. $x = 2$ is a property. Properties are relationships with no decorated variables.

6.3.1 Truth valued functions

A state relationship is a truth valued function and a list of arguments to that function. $2 = 4$ is a truth valued function which has the value false. $=$ is the function and 2 and 4 are the arguments. Assume we have a set of functions TVF...

$$TVF \triangleq \ldots \tag{6.16}$$

...which we do not enumerate here, but we assume the set includes such basic operators as $=$, $\leq$, $<$, *etc.* We also adopt pre-fix notation for simplicity. $2 = 4$ is expressed as $= \langle 2, 4 \rangle$. (This is just to make life easier whilst defining the semantics of our notations. The user of our notations should be able to express truth valued functions using in-fix or whatever notation suits him, so long as there is a clear and unambiguous mapping to pre-fix notation.)

The semantic function tv (standing for 'truth value') takes a function and a list of values as its arguments and returns the truth value of the function.

$$tv{:}TVF \times Val^* \rightarrow \mathbb{B} \tag{6.17}$$

As we have not enumerated TVF we cannot give a definition for tv, but it should be intuitive. For example...

$$tv[\![=, \langle 2, 4 \rangle]\!] \triangleq \mathsf{false}$$
$$tv[\![isEven, \langle 2 \rangle]\!] \triangleq \mathsf{true}$$
$$tv[\![\wedge, \langle \mathsf{true}, \mathsf{false} \rangle]\!] \triangleq \mathsf{false}$$
$$tv[\![isMortal, \langle \mathsf{Socrates} \rangle]\!] \triangleq \mathsf{true}$$

...and so forth.

6.3.2 Argument lists

This is all very well so long as we wish to evaluate truth valued functions over argument lists containing only constant values. We do not of course; also wish to use truth valued functions to describe the value of certain variables. In fact argument lists can contain three types of symbols

(collectively known as 'terms'); variable names (members of the set *Var*), constants (members of the set *Val*) and the special variable t denoting time.

$$Term \triangleq Var \cup Val \cup \{\mathrm{t}\} \tag{6.18}$$

However, to complicate matters each term in a relation's argument list may be decorated with certain symbols. A decoration is a symbol super-scripted to a term.

In RSSL we use four decorations; ^{n}, ^{p}, $'$ and ^{None} (an undecorated term *tm* is a short-hand for tm^{None}). The decorations used in RSSL are collected in the set *Dec*.

$$Dec \triangleq \{^{n}, ^{p}, ', ^{None}\} \tag{6.19}$$

A state relationship is a truth valued function and an argument list consisting of terms and their decorations.

$$R \triangleq TVF \times (Term \times Dec)^{*} \tag{6.20}$$

For example, assume that there is a truth valued function $EqInc : TVF$ (*EqInc* being short for 'equal to increment') which takes two arguments and holds true if the second argument is precisely one greater than the first.

$$\forall a, b : \mathbf{N} \bullet tv[\![EqInc, \langle a, b \rangle]\!] \triangleq a + 1 = b$$

The TLA action $x' = x + 1$ is a shorthand for the state relationship...

$$(EqInc, \langle (x, ^{None}), (x, ') \rangle)$$

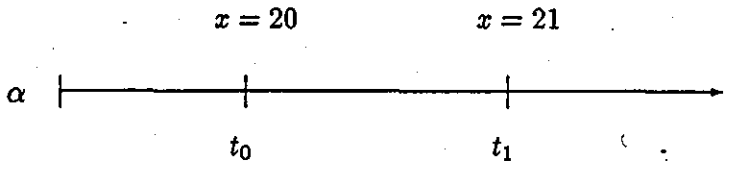


Figure 6-2: The TLA action to increment the value of x

6.3.3 Evaluating argument lists

To evaluate a state relationship we first need to evaluate its argument list into a list of values which we can then pass to the function tv. The value of a term is dependent on its decoration. Each decoration denotes a different state and time.

Consider figure 6-2. Given...

- an activity α,
- two states σ_0 and σ_1,
- two times t_0 and t_1, such that
- the value of x is 20 in σ_0 and 21 in σ_1, and
- $t_0 < t_1$

...then we can evaluate the TLA action $x' = x + 1$ in these states and times. If we evaluate undecorated terms using σ_0 and t_0 and the variables decorated with $'$ using σ_1 and t_1 then we can see that the action is true; it evaluates to $21 = 20 + 1$. (Time is dealt with implicitly and must advance; hence as t_0 is the time at the start of the action and t_1 the time at the end then $t_0 < t_1$.)

We define which state and time relate to which decoration in a state relationship with a 'decoration map'. This is a function from decoration to state and time.

$$DecMap \triangleq Dec \xrightarrow{p} (\Sigma \times \mathsf{T}) \tag{6.21}$$

(A decoration map is partial because in many circumstances we will be dealing with relationships that do not contain all decorations in their argument list.)

The decoration map we would use to evaluate the TLA action described above would be $\{^{None}\mapsto (\sigma_0, t_0), ' \mapsto (\sigma_1, t_1)\}$; undecorated variables are mapped to the state and time (σ_0, t_0) and variables decorated with $'$ are mapped to the state and time (σ_1, t_1).

The semantic function *eval* takes a term and a state/time pair and returns the value of the term in that state/time.

$$\begin{array}{l} eval: Term \to (\Sigma \times \mathsf{T}) \to Val \\ eval[\![x]\!](\sigma, t) \hat{=} \sigma(x) \text{ — where } x: Var \\ eval[\![c]\!](\sigma, t) \hat{=} c \text{ — where } c: Val \\ eval[\![\mathrm{t}]\!](\sigma, t) \hat{=} t \end{array} \tag{6.22}$$

In words; 'if the term is a variable then its value in σ is returned. If the term is a constant then the constant is returned. If the term is the special variable t then the time t is returned.'

6.3.4 Evaluating state relationships

Evaluating a state relationship is a matter of evaluating each of the decorated terms in its argument list against some decoration map, then passing these values to the function *tv* to determine the truth (or otherwise) of the relationship.

$$\begin{array}{l} rel: R \to DecMap \to \mathbb{B} \\ rel[\![f, \langle (m_1, d_1), \ldots, (m_n, d_n)\rangle]\!] dm \hat{=} \\ \qquad tv[\![f, eval[\![m_1]\!] dm(d_1), \ldots, eval[\![m_n]\!] dm(d_n)]\!] \end{array} \tag{6.23}$$

In words; 'each term from m_1 to m_n is evaluated by extracting the state and time from the decoration map *dm* appropriate to the decoration (d_1 to d_n) of the term. This evaluated argument list is then passed to the function *tv*.'

As an example let us work through the TLA action $x' = x + 1$. We have already shown that $x' = x + 1$ is a shorthand for the state relationship $(EqInc, \langle (x, ^{None}), (x, ')\rangle)$. Let us evaluate this against the states and times shown in figure 6-2. The decoration map for this evaluation is $dm = \{^{None}\mapsto (\sigma_0, t_0), ' \mapsto (\sigma_1, t_1)\}$ where $\sigma_0(x) = 20$ and $\sigma_1(x) = 21$.

We are then evaluating the following formula...

$$rel[\![EqInc, \langle (x, ^{None}), (x, ')\rangle]\!] dm$$

...which is equal by definition to...

$$tv[\![EqInc\langle eval[\![x]\!]dm(^{None}), eval[\![x]\!]dm(')\rangle]\!]$$

Now we extract the appropriate states and times from the decoration map dm...

$$tv[\![EqInc, \langle eval, [\![x]\!](\sigma_0, t_0), eval[\![x]\!](\sigma_1, t_1)\rangle]\!]$$

...and we can then evaluate the two terms against these states and times...

$$tv[\![EqInc\langle 20, 21\rangle]\!]$$

...and finally we pass this list of values to the function tv to get the result...

true

6.3.5 Properties

A property describes what is true at a single state and time. Its definition is very similar to that given for state relationships.

A property is a truth valued function and sequence of (undecorated) terms.

$$P \hat{=} TVF \times Term^* \tag{6.24}$$

The semantic function for properties *prop* evaluates a property given a state and time.

$$\begin{aligned} &prop : P \to (\Sigma \times \mathsf{T}) \to \mathbb{B} \\ &prop[\![f, \langle m_1, \ldots, m_n\rangle]\!](\sigma, t) \hat{=} \\ &\qquad tv[\![f, \langle eval[\![m_1]\!](\sigma, t), \ldots, eval[\![m_n]\!](\sigma, t)\rangle]\!] \end{aligned} \tag{6.25}$$

In words; 'each term in the argument list of the property is evaluated against (σ, t) and this evaluated list is passed to the function tv.'

6.4 A simple temporal logic

We have now put down enough formal preliminaries that we can begin to formalise RSSL's temporal logic notation. Recall from the introductory section that we need to define the model of system behaviour (which we did in section 6.2), the syntax of the notation and the semantic function for the notation.

6.4.1 Formulae

The syntax for the temporal logic recursively defines a set of 'formulae' F. A formula is a well formed temporal logic expression.

The base case for the syntax definition is that all properties as defined in section 6.3.5 are formulae. If x is a variable name, δ is a time and ϕ and ψ are formulae then so are...

$$\neg\phi \qquad \phi \wedge \psi \qquad \exists x \bullet \phi \qquad \Box\phi \qquad \Diamond\phi \qquad \Diamond_\delta \phi$$

...and nothing else is a formula.

Formally...

$$\begin{array}{l} F ::= P \quad | \\ \quad \neg F \mid F \wedge F \mid \exists Var \bullet F \mid \\ \quad \Box F \mid \Diamond F \mid \Diamond_{\mathsf{T}} F \end{array} \tag{6.26}$$

The operators $\neg$, $\wedge$ and $\exists$ have their usual predicate logic meanings; negation, conjunction and existential quantification respectively. The temporal operator $\Box$ has the meaning 'henceforth', the operator $\Diamond$ is 'eventually' and $\Diamond_{\mathsf{T}}$ is 'timed-eventually'.

6.4.2 Semantics for formulae

We begin the semantics for formulae by defining the semantic formulae *form* which returns whether a given formula holds true at a given time in a given activity. The given time is assumed to be 'now'. So if the current time is ten, the formula $x = 15 \wedge \mathit{isMortal}(y)$ holds true in activity α if $\alpha(10)(x) = 15$ and $\alpha(10)(y) = \mathsf{Socrates}$.

The semantic function *form* has the following signature...

$$form : F \to (Activity \times \mathsf{T}) \to \mathbb{B} \tag{6.27}$$

The base case — properties

form for the base case formulae (*i.e.* properties) is based on the function *prop* (equation 6.25 in the previous section). We use the given time to extract the current state from the given activity (using the double brackets notation — see formula 6.10) and then pass this to the *prop* function to evaluate the property. Given that $pr : P \ldots$

$$form[\![pr]\!](\alpha, now) \triangleq prop[\![pr]\!]\alpha((t)) \tag{6.28}$$

Negation and conjunction

The definitions for $\neg$ and $\wedge$ are straight forward and should be familiar to a reader with any experience with denotational semantics. For the formula $\neg\phi$ (where ϕ is a formula) to hold now in an activity then it is the case that ϕ does not hold now in the activity. Similarly for $\phi \wedge \psi$ (where ϕ and ψ are both formulae) to hold now in an activity it is the case that both ϕ and ψ hold in the activity.

Formally, given that ϕ and ψ are formulae...

$$form[\![\neg\phi]\!](\alpha, now) \triangleq \neg form[\![\phi]\!](\alpha, now) \tag{6.29}$$

$$form[\![\phi \wedge \psi]\!](\alpha, now) \triangleq form[\![\phi]\!](\alpha, now) \wedge form[\![\psi]\!](\alpha, now) \tag{6.30}$$

Existential quantification

To define existential quantification we use the *variant* equivalence (equation 6.11) we defined on activities. For $\exists x \bullet \phi$ (where x is a variable and ϕ is a formula) to hold now in a given activity it is the case that there is at least one related activity in which ϕ holds now. The related activity should be the same as the given activity except (possibly) for its values of x. In other words

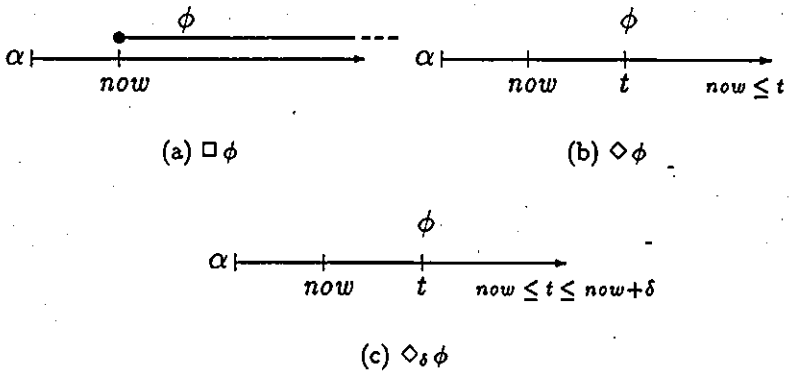


Figure 6-3: Pictorial representations of the temporal logic operators

there exist some values for x (not necessarily those in the given activity) such that ϕ holds.

Formally, given that x is a variable name and ϕ is a formula...

$$form[\![\exists x \bullet \phi]\!](\alpha, now) \triangleq \exists \alpha' : Activity \bullet variant(\alpha, \alpha', x) \wedge form[\![\phi]\!](\alpha', now) \tag{6.31}$$

The henceforth operator

For $\Box\,\phi$ (where ϕ is formula) to hold now in an activity ϕ must hold now and at all times in the future. $\Box\,\phi$ is graphically illustrated in figure 6-3(a).

Formally, given that ϕ is a formula...

$$form[\![\Box\,\phi]\!](\alpha, now) \triangleq \forall t : \mathsf{T} \bullet t \geq now \Rightarrow form[\![\phi]\!](\alpha, t) \tag{6.32}$$

In words; 'ϕ holds true in α at all times after (and including) *now*'.

The eventually operators

The eventually operator $\Diamond$ (illustrated in figure 6-3(b)) states that a formula holds now or at some time in the future.

$$form[\![\Diamond\,\phi]\!](\alpha, now) \triangleq \exists t : \mathsf{T} \bullet t \geq now \wedge form[\![\phi]\!](\alpha, t) \tag{6.33}$$

In words; 'there is a time t which is later than (or equal to) *now* at which ϕ holds true in α.'

The eventually operator can also be expressed as a dual of the always operator (and *vice versa*). Hence...

$$\Diamond \phi \triangleq \neg \Box \neg \phi \tag{6.34}$$

In words; 'it is not the case that ϕ never happens.'

The timed eventually operator (illustrated in figure 6-3(c)) places a limit on the amount of time it takes for the formula to become true. The time limit is sub-scripted to the eventually operator.

$$form[\![\Diamond_\delta \phi]\!](\alpha, now) \triangleq \exists t : \mathsf{T} \bullet now \leq t \leq now + \delta \wedge form[\![\phi]\!](\alpha, t) \tag{6.35}$$

In words; 'there is a time t which is later than (or equal to) *now* but not later than $now + \delta$ at which ϕ holds true in α.'

So far we have described a fairly sparse set of operators for our temporal logic. We can now proceed to show several definitions of other operators as syntactic replacements of the operators we have already defined.

Other predicate logic operators

Predicate logic operators are defined in the traditional way...

$$\phi \vee \psi \triangleq \neg(\neg\phi \wedge \neg\psi) \tag{6.36}$$

Disjunction

$$\phi \Rightarrow \psi \triangleq \neg\phi \vee \psi \tag{6.37}$$

Implication

$$\phi \Leftrightarrow \psi \triangleq (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \tag{6.38}$$

Biconditional

$$\forall x \bullet \phi \triangleq \neg \exists x \bullet \neg\phi \tag{6.39}$$

Universal quantification

Followed by

A weak idea of sequencing is introduced using the $\rightsquigarrow$ operator. $\phi \rightsquigarrow \psi$ means that whenever ϕ holds true it is followed sometime in the future by ψ holding true.

$$\phi \rightsquigarrow \psi \triangleq \Box(\phi \Rightarrow \Diamond \psi) \tag{6.40}$$

Note that this operator does *not* state that ψ being true is *immediately* followed by ψ being true. This operator should not be therefore used as a replacement for sequencing as normally used in the programming language sense.

Other temporal logic operators

Pnueli [98] defines a clutch of other temporal logic operators, including until, waiting and 'strict' variants of the henceforth and eventually operators. The strict operators state that something holds at times *not* including now. Usually temporal logics give a semantic definition for an until operator and define henceforth and eventually from it. We have limited ourselves to defining just the operators needed to define safety and liveness properties. If needed however we are confident we can define the other operators defined by Pnueli. (Lamport [74] argues that only the bare minimum of temporal operators should be used as they tend to be unfamiliar and cause confusion. We broadly agree with this argument.)

Many temporal logics include the 'next' operator $\bigcirc$. In a real time model 'next' has little meaning; a given time has no defined 'next' time. We could however define $\bigcirc \phi$ to have the meaning 'ϕ holds after the next state change'.

But we still have philosophical difficulties with this definition. Assume we wish to say that after the next state change x has the value 2; $\bigcirc(x = 2)$. What if $x = 2$ already holds in the current state?

We omit a definition of $\bigcirc$ because we believe that in the context of a real time system it can very easy be misused by the unwary.

6.4.3 The behaviour described by a formula

The behaviour described by a formula is the set of all activities that satisfy the formula from time zero. Hence sub-formulae that are not guarded by a temporal logic operator hold true at time zero.

$$\begin{array}{l} f : F \rightarrow behaviour \\ f[\![\phi]\!] \triangleq \{\alpha : Activity \mid form[\![\phi]\!](\alpha, 0)\} \end{array} \tag{6.41}$$

A definition of the semantic function f is the purpose of this section; given any temporal logic formula ϕ we can apply f to calculate the behaviour of a system for which ϕ is a statement of requirements.

For example the formula $x = 0 \wedge \Box(x \leq x')$ describes a system where the value of x starts at 0 and never decreases. $f[\![x = 0 \wedge \Box(x \leq x')]\!]$ defines the set of all activities which start with x at 0 and continue (to infinite time) with x never decreasing. (Note that this is a safety condition; x is not guaranteed to increase in this formula.)

So in this section, as desired, we have defined the set of all simple temporal logic descriptions (equation 6.26) and defined how those descriptions relate to behaviour (equation 6.41).

6.5 Computations

Essentially an RSSL specification describes how a collection of its unit of functionality, a computation, is legally performed by a system. Before being able to give the semantics for RSSL specifications we first need to thoroughly discuss and define computations.

TLA uses an 'action' as its unit of functionality. An action is simply a relationship between two states. We take a rather more operational entity as our unit of functionality which we call a 'computation'. (Computations were briefly introduced in section 5.5.1.)

The simplicity of an action means that it is a very powerful descriptive tool, but this also means that it can be easily misused. Our computation is a specialisation of an action which we consider to be 'sensible'. Computations are therefore more constrained, but, we hope, not restrictively so.

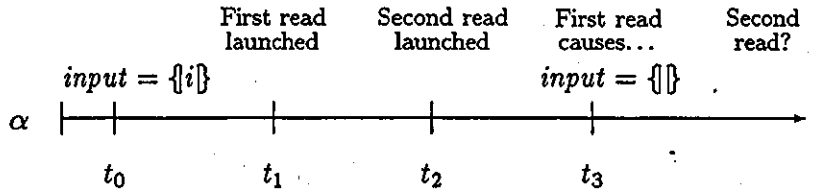


Figure 6-4: The problem with launching concurrent actions

6.5.1 The problem with actions

To show a possible difficulty with actions consider the following example. In a multiprocessor system we wish to read items from an input bag. The action *read* takes items from the input bag *input*. In TLA we could specify *read* as...

$$input \neq \emptyset \wedge input = \{|x'|\} \uplus input'$$

The above action is 'enabled' when $input \neq \emptyset$ — the input bag is not empty, or in other words there are items in the input to be read. The action takes an item from the input bag and places the value on x. (Presumably there will be some other processing that will do something to x, but this is not our concern here.)

Assume now that we can launch several instances of *read* concurrently. Obviously we cannot launch any instances until the action is enabled, but what if there is just one item in the input list and we concurrently launch more than one instance of *read*? The first instance to actually remove the item from the input bag will disable the other instances, so they cannot complete, or will complete with undefined results. We wish to prevent this sort of interference.

This problem is illustrated in figure 6-4. At time t_0 there is one item in the input bag and so at time t_1 an instance of *read* is launched. However, for some reason that action is slow at getting going and at t_2 the item is still in the input list and so another instance of *read* is launched. At t_3 the first instance of read finally removes the item from the input bag and this leaves the second instance hanging; launched, but with no items to read from the input bag.

Typically concurrent systems will have locking mechanisms on critical sections, so that only one instance can access the data at a time, or we could only launch one instance at a time. Lock-

ing mechanisms are rather concrete; we wish to be more abstract about our systems. Performing one action at a time reduces us to sequential systems.

The pseudo-interleaved concurrent model

We overcome these problems by assuming that we have an 'pseudo-interleaved concurrent' model of system behaviour. We assert that typically actions will read data from some 'public' space to some 'private' space, perform processing on the data in the private space and then write the result of the processing onto the public space. Because the private space is internal to each instance of an action then we can perform processing on it without interfering with any other concurrently working actions (or, indeed, being interfered with by other actions). It is only when reading and writing to the public space that interference can occur. We therefore 'interleave' the reads and writes; only one can occur at a time, but the processing can occur concurrently.

We refer to this approach as *pseudo*-interleaved because although the semantics describe only this interleaved behaviour we wish to be able to generalise this interleaving to fully concurrent systems. The argument being that we wish to implement systems that have the *same effect* as the behaviour described by their specifications. The semantics we shall define in the next chapter will be a *canonical* model based on this pseudo-interleaved model. We could develop this model into a truly concurrent model using the approaches shown in [7] *etc.*

Justifying computations

A computation is a unit of functionality that makes the read, write and process phases explicit. This at first sight is rather complicated and limiting. We are apparently dictating an architecture for our language by stating that all actions act in a certain way. Such a conceptual overhead surely limits how abstract our language can be.

We contend, however, that dictating that a unit of functionality behaves in (what we believe to be) a sensible way greatly eases much of the reasoning we would wish to do with our models. For example, following from Abadi and Lamport [1] we would wish that if we had n subsystems specified by the formulae $\phi_1, \ldots, \phi_n$ then the specification of the overall system ϕ would be $\phi_1 \wedge \ldots \wedge \phi_n$. Things are not that simple, of course. Abadi and Lamport give theorems which specifications must fulfill for the above simple conjunction to hold. Although a great

improvement on previous techniques the theorems are still rather involved and fiddly. The problem being that $\phi = \phi_1 \wedge \ldots \wedge \phi_n$ breaks down when the sub-systems start interfering with each other.

Using computations limits this interference and so the decomposition of specifications becomes rather simpler. Furthermore, we contend that computations are a natural way of thinking about systems and they have plenty of precedent in the hardware and concurrency literature. Prescribing a system as being based on our notion of computations should not therefore be too much of a radical departure from the norm for system designers.

6.5.2 A syntax for computations

Recall from the previous chapter we introduced computations in a form with five clauses; this was the 'shorthand' form, where the read, process and write phases are expressed in one single relationship. For more complicated computations it will be preferable to describe the read, process and write phases separately. This is done with the 'longhand' form. The two forms only differ in style — they are mathematically equivalent. Clarity to the reader should be paramount; simple, elegant computations are probably best expressed in the shorthand form so as not to be cluttered with (probably trivial) relationships in the private space. More complex, concrete computations are probably better suited to the longhand form.

Longhand computations

Longhand computations describe computations in much the same way as we characterised them earlier in this section.

$$\begin{aligned} longC \mathrel{\hat{=}} & \textbf{input} : X \\ & \textbf{output} : Y \\ & \textbf{private} : I \\ & \textbf{enabled\_by} : E \\ & \textbf{read} : Re \\ & \textbf{side-effect} : S \\ & \textbf{process} : Pr \\ & \textbf{write} : W \end{aligned} \tag{6.42}$$

...where...

- $X : \mathcal{F}(Var)$ is a finite set of variables which are public (shared by other computations) and are read by the computation,
- $Y : \mathcal{F}(Var)$ is a finite set of variables which are public (shared by other computations) and are written to by the computation,
- $I : \mathcal{F}(Var)$ is a finite set of private variables,
- $E : P$ is an enabling condition,
- $Re : R$ is a state relationship (containing variables from X which are undecorated and variables from I which are decorated with ^{n}) describing the how values are copied from the public space to the private,
- $S : R$ is a relationship (only containing variables from X which are undecorated or decorated with ^{n}) describing the side-effect; how the public space is effected by the read phase,
- $Pr : R$ is a relationship (only containing variables from I decorated by ^{n} or ^{p}) describing how the private space is updated by the processing phase,
- W is a relationship (containing variables from I decorated by ^{p} and variables from Y decorated by ^{p} or $'$) describing how the public space is updated in the write phase, usually by copying values from the private space,
- if the enabling predicate is omitted it is assumed to be the logical constant **true**,
- if any of the relationship clauses are omitted they are assumed to be the identity,
- if quantified variables are introduced in any of the clauses then the scope of those variables carries down to all the following clauses (unless explicitly prevented by bracketing).

Note that although the input, output and internal clauses denote a set of variable names, we need not bother with the delimiting set brackets in the actual declaration.

$$\textbf{input} : \{a, b, c\}$$

...is the same as...

$$\textbf{input} : a, b, c$$

Shorthand computations

Shorthand computations describe relationships between variables in the public scope only.

$$\begin{aligned} shortC \triangleq\ & \textbf{input} : X \\ & \textbf{output} : Y \\ & \textbf{enabled\_by} : E \\ & \textbf{side-effect} : S \\ & \textbf{outcome} : O \end{aligned} \tag{6.43}$$

...where...

- $X : \mathcal{F}(Var)$ is a finite set of public variables that are read by the computation,
- $Y : \mathcal{F}(Var)$ is a finite set of public variables that are written to by the computation,
- $E : \dot{P}$ is an enabling condition,
- $S : R$ is a relationship (only containing variables from X which are undecorated or decorated with ^{n}) describing the side-effect; how the public space is effected by the read phase,
- $O : R$ is a state relationship (containing undecorated variables from X and variables from Y decorated by ^{p} or $'$) describing the relationship between the public space in the launch state, the penultimate state and the final state, and
- the same rules concerning omission of the enabling predicate and relations and the scope of quantified variables that applied to longhand computations apply to shorthand computations.

Are long and short-hand computations equivalent?

In order to make the link between the long and short-hand forms clear, assume that the relationships describe functions on the state. A generic long-hand computation would have the following form...

$$
\begin{aligned}
\mathit{genericLongC} \mathrel{\hat{=}} & \textbf{input} : x \\
& \textbf{output} : y \\
& \textbf{private} : i \\
& \textbf{enabled\_by} : E \\
& \textbf{read} : i^n = R(x) \\
& \textbf{side-effect} : x^n = S(x) \\
& \textbf{process} : i^p = P(i^n) \\
& \textbf{write} : y' = W(i^p, y^p)
\end{aligned}
\tag{6.44}
$$

The following short-hand computation is equivalent to the above computation.

$$
\begin{aligned}
\mathit{equivC} \mathrel{\hat{=}} & \textbf{input} : x \\
& \textbf{output} : y \\
& \textbf{enabled\_by} : E \\
& \textbf{side-effect} : x^n = S(x) \\
& \textbf{outcome} : \exists i^n, i^p \bullet i^n = R(x) \wedge \\
& \qquad\qquad\qquad\qquad i^p = P(i^n) \wedge \\
& \qquad\qquad\qquad\qquad y' = W(i^p, y^p)
\end{aligned}
\tag{6.45}
$$

The internal variables are existentially quantified and the the outcome is the composition of R, P and W.

By substituting appropriate functions into formula 6.44 we can describe computations and for any long-hand computation in this form we can derive an equivalent short-hand computation (as shown in formula 6.45), hence computations can be expressed equivalently in long or short-hand.

6.5.3 An abstract syntax for computations

As the two forms are equivalent the abstract syntax of computations is based on the shorthand form for simplicity. A computation is a 5-tuple...

$$C \mathrel{\hat{=}} (\mathcal{F}(\mathit{Var}) \times \mathcal{F}(\mathit{Var}) \times P \times R \times R) \tag{6.46}$$

...where iff $(X, Y, E, S, O) : C$ then...

- X is the input space of the computation,
- Y is the output space of the computation,

- E is the enabling predicate,
- S is the side-effect, and
- O is the outcome.

There are rather involved well-formed conditions for computations which we shall discuss and formalise in the next section when we give a full semantics for them.

6.5.4 The semantics for computations

A computation is semantically characterised by three functions;

- the *enable* function defines the state/times in which a computation is enabled,
- the *side* function is given an initial state/time and defines a set of acceptable next state/times signifying the completion of the read phase, and
- the *outcome* function is given initial and penultimate state/times and defines a set of acceptable final state/times signifying the completion of the computation.
- the *do* function combines the above three functions to describe a legal performance of a computation over four state/time pairs.

Semantics for the enabling predicate

We have previously given semantics for properties using the *prop* function. *enable* uses the *prop* function to define a set of state/times in which a computation is enabled.

$$\begin{array}{l} enable : C \to \mathcal{P}(\Sigma \times \mathsf{T}) \\ enable[\![(X, Y, E, S, O)]\!] \triangleq \{(\sigma, t) : \Sigma \times \mathsf{T} \mid prop[\![E]\!](\sigma, t)\} \end{array} \tag{6.47}$$

In words; 'the set of all state/times in which the E property holds are returned.'

Semantics for the side-effect

In the state relationship that describes the side-effect undecorated arguments denote values in the 'launch' state/time and arguments decorated by ^{n} denote values in the 'next' state/time (recall

figure 5-4). Therefore in order to define the semantics for the side-effect we pass a decoration map that maps the decoration *None* to the launch state/time and ^{n} to the next state/time and pass this to the semantic function for state relationships *rel* to evaluate the side-effect.

$$\begin{array}{l} side : C \to (\Sigma \times \mathsf{T}) \to \mathcal{P}(\Sigma \times \mathsf{T}) \\ side[\![(X, Y, E, S, O)]\!](\sigma, t) \triangleq \\ \qquad \{(\sigma^n, t^n) : \Sigma \times \mathsf{T} \mid rel[\![S]\!]\{^{None}\!\mapsto (\sigma, t), ^n \mapsto (\sigma^n, t^n)\} \wedge \\ \qquad\qquad canVary(\sigma, \sigma^n, X)\} \end{array} \tag{6.48}$$

In words; 'the set of all 'next' state/time pairs (σ^n and t^n) which are valid next state/times in relation to the given computation and the given 'launch' state/time are returned. The *canVary* predicate ensures that the only variables that are altered in the side-effect are those defined by the input clause to be accessible to the computation to read.'

If the input space is $\{x\}$ and the side-effect is $x^n = x + 1$ then x is incremented by the side-effect and all other public variables are unchanged. More subtly if the input space is $\{x, y\}$ and the side-effect is $x^n = x + 1$ then x is incremented by the side-effect and y can assume any value. If we do not want y to change in the side-effect, we must state this explicitly; $x^n = x + 1 \wedge y^n = y$.

Semantics for the outcome

The semantic function for the outcome relation is similar to that for the side-effect. A decoration map that maps the appropriate decoration to state/time pair and the outcome relationship is evaluated using the *rel* function. Again the *canVary* predicate ensures that only those variables in the output space are altered by the outcome.

$$\begin{array}{l} outcome : C \to (\Sigma \times \mathsf{T}) \to (\Sigma \times \mathsf{T}) \to \mathcal{P}(\Sigma \times \mathsf{T}) \\ outcome[\![(X, Y, E, S, O)]\!](\sigma, t)(\sigma^p, t^p) \triangleq \\ \qquad \{(\sigma', t') : \Sigma \times \mathsf{T} \mid rel[\![O]\!]\{^{None}\!\mapsto (\sigma, t), ^p \mapsto (\sigma^p, t^p), ' \mapsto (\sigma', t')\} \wedge \\ \qquad\qquad canVary(\sigma^p, \sigma', Y)\} \end{array} \tag{6.49}$$

In words; 'given a computation, launch state/time and penultimate state/time the set of all valid final state/times is returned. The decoration ^{None} is mapped to the start state/time, the decoration ^{p} to the penultimate state/time and $'$ to the final state/time. The relationship O is evaluated using this decoration map and *canVary* assures that only the variables in the output

space are changed by the outcome.'

Putting the enabling condition, the side-effect relation and the outcome relation together

We can now define a predicate *do* which describes the performance of a computation over four state/time pairs.

$$\begin{aligned} &do : C \to (\Sigma \times \mathsf{T})^4 \to \mathbb{B} \\ &do[\![c]\!]((\sigma,t),(\sigma^n,t^n),(\sigma^p,t^p),(\sigma',t')) \triangleq (\sigma,t) \in enabled[\![c]\!] \wedge \\ &\qquad\qquad (\sigma^n,t^n) \in side[\![c]\!](\sigma,t) \wedge \\ &\qquad\qquad (\sigma',t') \in outcome[\![c]\!](\sigma,t)(\sigma^p,t^p) \wedge \\ &\qquad\qquad t < t^n \leq t^p < t' \end{aligned} \tag{6.50}$$

In words; 'the four state/times are a valid performance of a computation c if...

- c is enabled in the launch state/time (σ,t),
- the step from the launch state/time to the next state/time, (σ,t) to (σ^n,t^n), is a valid side-effect for c, and
- the step from the penultimate state/time to the final state/time, (σ^p,t^p) to (σ',t'), is a valid outcome (with respect to the launch state/time), and
- the times are correctly ordered.'

Well formed conditions for computations

We glanced over the conditions for computations to be well formed when we gave their syntax because at that point we did not have the apparatus to deal with them. We can define them now using their semantic definitions.

The enabling condition should imply the side-effect and outcome relations; in other words for every state/time the computation is enabled in there must be at least one resultant state defined for the side-effect and outcome. We say the side-effect and outcome must be 'wider' than the enabling condition. Formally...

$$
\begin{aligned}
&\mathit{wider}:C \rightarrow \mathbb{B} \\
&\mathit{wider}(c) \triangleq \mathit{enable}[\![c]\!] \subseteq \{(\sigma,t) \mid \mathit{side}[\![c]\!](\sigma,t) \neq \emptyset\} \wedge \\
&\qquad\qquad \mathit{enable}[\![c]\!] \subseteq \{(\sigma,t) \mid \exists(\sigma^p,t^p) \bullet \mathit{outcome}[\![c]\!](\sigma,t)(\sigma^p,t^p) \neq \emptyset\}
\end{aligned} \tag{6.51}
$$

In words; 'the first line of the definition of *wider* states that the set of all state/times in which a computation is enabled is a sub-set of the set of all launch state/time pairs for which there is a next state/time pair defined. In a similar way the second line states that the enabled set is a sub-set of all launch state/time pairs for which there is a final state/time defined.'

A second condition states that it is always the case that it is possible for the write phase to eventually occur. Unfortunately this 'well formed' condition is dependent on the context of the specification.

Consider the following system; some environment computation periodically places values to be processed on an input variable *input*. *input* is either a natural number or the token Null.

$$
\mathit{input}:\mathbb{N} \cup \{\mathsf{Null}\} \tag{6.52}
$$

If there is a natural number on *input* then we assume that that number is yet to be processed. If Null is on *input* then we assume that there is no data to be processed. The environment computation effectively changes *input* from Null.

A kernel computation is enabled when there is a natural number on *input*, it takes the natural number off *input* processes it using some function f and places the result on the end of an output list.

$$
\mathit{output}:\mathbb{N}^* \tag{6.53}
$$

Hence...

$$
\begin{aligned}
\mathit{kernel} \triangleq\ & \textbf{input} : \mathit{input} \\
& \textbf{output} : \mathit{output} \\
& \textbf{enabled\_by} : \mathit{input} \neq \mathsf{Null} \\
& \textbf{side-effect} : \mathit{input}^n = \mathsf{Null} \\
& \textbf{outcome} : \mathit{output}' = \mathit{output}^p \frown \langle f(\mathit{input}) \rangle
\end{aligned} \tag{6.54}
$$

This is all very well, but what if we limit the size of the output sequence to (say) 10? We cannot include $|output| < 10$ in the enabling condition because a computation may take an input value when the output list is not full, take a long time over processing it and find that several other computations have 'overtaken' it and filled up the output list.

The problem is that it may not be possible to predict at the time of launching a computation whether or not it will be possible to perform the write phase. Indeed we can envisage computations that can launch at times when it is *impossible* to perform the write phase, but we are assured that something will happen in the system to make the write phase become possible.

So the computation *kernel* working on a limited output list is not invalid *per se*, but we need to be aware of the rest of the system (maybe the environment will not place more than 10 numbers on *input* anyway, or there may be another kernel computation that takes numbers out of the output list, so even if it fills up it will eventually empty again).

We can define a set of 'write complete' computations, for which the write phase is possible in all states and times.

$$\begin{aligned} &wComplete : C \to \mathbb{B} \\ &wComplete(c) \triangleq (\Sigma \times \mathsf{T}) = \{(\sigma^p, \mathrm{t}^p) \mid \exists (\sigma, t) : (\Sigma \times \mathsf{T}) \bullet \\ &\qquad\qquad outcome[\![c]\!](\sigma, t)(\sigma^p, t^p) \neq \emptyset\} \end{aligned} \tag{6.55}$$

In words 'the set of penultimate state/time pairs for which there is a final state/time defined is the set of *all* state/time pairs.'

Making all the computations in a specification write complete is a very strong requirement, but it assures that we do not run into the sort of problems described above.

6.6 The syntax for the RSSL specification notation

A system specification consists of a property which describes acceptable initial states for the system and a collection of computations which determine how the system evolves from an initial state. In certain situations computations become enabled and may then be launched; the specification includes deontic operators which assert whether a computation may or must be launched.

6.6.1 Specification syntax

A specification has the following syntax...

$$init \wedge \Box[\bigvee_{i \in 1\,..\,n} k_i] \wedge \Box\langle \bigvee_{i \in 1\,..\,m} e_i \rangle \tag{6.56}$$

...where...

- *init* is a property,
- $\{k_1, \ldots, k_n\}$ is a set of kernel computations, and
- $\{e_1, \ldots, e_m\}$ is a set of environment computations.

init is the initial property, which can describe any configuration of the state but must describe time as being zero. The operator $\Box$ has its temporal meaning; 'henceforth', square brackets round a disjunction of computations means that the disjunction is obligatory, angle brackets mean the disjunction is optional.

6.6.2 Abstract syntax for specifications

An abstract syntax for a system specification is a 3-tuple...

$$RSSL \triangleq (P \times \mathcal{F}(C) \times \mathcal{F}(C)) \tag{6.57}$$

...where if $(init, obl, opt) : RSSL$ then ...

- *init* is the initial property, which describes states at time zero,
- *obl* is the set of obligatory computations, and
- *opt* is the set of optional computations.

6.7 The semantics for the RSSL specification notation

Recall that the result of section 6.4 was the semantic function f that given a temporal logic formula returns the set of all activities (the behaviour) described by that formula. The purpose

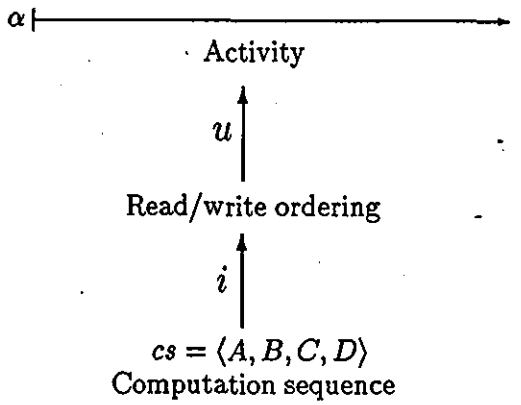


Figure 6-5: An overview of the approach to defining RSSL specification semantics

of this section is the same; the definition of a semantic formula s which takes a specification and returns the behaviour described by the specification.

$$s{:}RSSL \rightarrow Behaviour$$

However specifications are rather more involved than temporal logic formulae and so the route we take to a definition of s will be rather more circuitous.

6.7.1 An overview of the approach to be taken

There are three 'levels' to the semantic definition of RSSL specifications — the 'computation sequence' level, the 'read/write ordering' level and the 'activity' level. (See figure 6-5).

We start with a sequence of computations. The middle level is the interleaving of the read and write phases of those computations. For each computation in the computation sequence its read and write phase should occur in the read/write ordering, its read phase should occur before its write phase and read phases should occur in the same order as their computations do in the computation sequence level. The final level is an activity which contains (only) the state changes caused by the read and write phases in the read/write ordering. There are functions from one level to the next — the function from the computation sequence to the read/write ordering is denoted i (for 'interleaving') and the function from the read/write ordering to an

activity is denoted u (for 'updates').

The rest of this section concerns the definition of those functions and how they are used to describe the set of all activities that are legal with respect to a specification.

6.7.2 Interleaving functions that take computation sequences to read/write orderings

I is the set of all interleaving functions.

$$I \mathrel{\hat{=}} \mathbb{N}_1 \rightarrow (\mathbb{N}_1 \times \mathbb{N}_1) \tag{6.58}$$

A function $i : I$ projects the ordinal position of computations in the computation sequence to the ordinal position of their read and write phases in the read/write ordering.

If a computation is the nth to occur in the computation sequence and $i(n) = (x, y)$ then its read phase occurs xth and its write phase occurs yth in the read/write ordering. A useful syntactic sugar separates the read and write parts in the range of a function i as follows...

$$\begin{array}{l} \forall i : I \bullet \\ \quad i_r = \{(n \mapsto x) \mid \exists y : \mathbb{N}_1 \bullet (n \mapsto (x, y)) \in i\} \wedge \\ \quad i_w = \{(n \mapsto y) \mid \exists x : \mathbb{N}_1 \bullet (n \mapsto (x, y)) \in i\} \end{array} \tag{6.59}$$

In words; 'for all functions i there are two related functions $i_r : \mathbb{N}_1 \rightarrow \mathbb{N}_1$ and $i_w : \mathbb{N}_1 \rightarrow \mathbb{N}_1$ which return the ordinal positions of read phases and write phases respectively.' Hence if a computation occurs nth in the computation sequence then its read phase occurs $i_r(n)$th in the read/write ordering and its write phase occurs $i_w(n)$th.

An interleaving function must fulfill several properties to correctly map a computation sequence to a read/write ordering. We define a predicate *interleaving* that takes a function i and holds true if i has the properties we require.

$$\begin{array}{l} \mathit{interleaving} : I \rightarrow \mathbb{B} \\ \mathit{interleaving}(i) \mathrel{\hat{=}} \\ \quad 1. \quad \mathit{contig}(\mathbf{dom}\ i) \wedge \\ \quad 2. \quad \mathit{contig}(\mathbf{ran}\ i_r \cup \mathbf{ran}\ i_w) \wedge (\mathbf{ran}\ i_r \cap \mathbf{ran}\ i_w = \emptyset) \wedge \\ \quad\ \ \forall n, n' : \mathbb{N}_1 \bullet \{n, n'\} \subseteq \mathbf{dom}\ i \Rightarrow \\ \quad 3. \quad (i_r(1) = 1 \wedge n > n') \Rightarrow (i_r(n) > i_r(n')) \wedge \\ \quad 4. \quad i_w(n) > i_r(n) \end{array} \tag{6.60}$$

In words; 'the numbers in the left hand column refer to the enumerated points below. For a function i to be interleaving, it must...

1. have a contiguous domain. (See B.9 for a formal definition of the predicate *contig.*) *i.e.* if the domain is finite and the number n is in the domain then $1..n$ must be in the domain, if the domain is infinite then all non-zero natural numbers must be included in the domain. This ensures that if (say) there are four computations in the computation sequence then there are read and write positions defined for the first, second, third and fourth computations.

2. have a contiguous and non-overlapping range. This means that if the domain of i is $\{1..n\}$ then the union of the ranges of i_r and i_w is $\{1..2n\}$. This ensures that there cannot be a point in the read/write ordering (other than those greater than $2n$) for which nothing happens and that only one read or[2] write happens at each ordinal point in the read/write ordering.

3. result in a function i_r that is strictly increasing from 1. This means that the read phase of the first computation in the computation sequence is the first thing to happen and the computations in the computation sequence happen in the same order as their read phases in the read/write ordering.

4. result in functions i_r and i_w such that the read phase of a computation always occurs prior to its write phase.'

6.7.3 Update functions that take read/write orderings to activities

An activity is a series of state changes. A specification should account for all of those state changes, *i.e.* an activity that is correct with regard to a specification has all its state changes predicted by the specification. The function u maps the read and write phases of computations to state changes in an activity.

[2]Exclusive or!

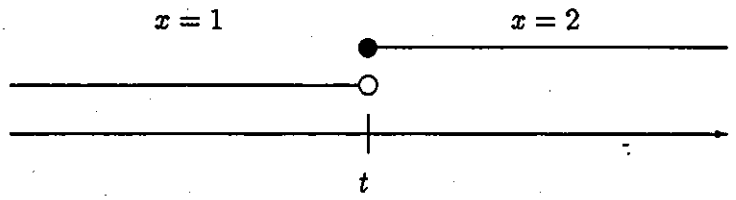


Figure 6-6: A state discontinuity shown pictorially

State discontinuities

First we must capture 'state discontinuities'. A discontinuity is the moment in time at which the state changes. For a time t to be a discontinuity then the state at the times immediately earlier than t must be different to that state at t. Consider figure 6-6. Time t is the precise point that the variable x changes from 1 to 2.

The predicate *discont* holds true if a given time is a change point in a given activity.

$$\begin{aligned} &discont : Activity \times \mathsf{T} \to \mathbb{B} \\ &discont(\alpha, t) \mathrel{\hat{=}} \exists early : \mathsf{T} \bullet early < t \wedge \alpha(early) \neq \alpha(t) \wedge \\ &\qquad\qquad \forall mid : \mathsf{T} \bullet early \leq mid < t \Rightarrow \alpha(early) = \alpha(mid) \end{aligned} \tag{6.61}$$

In words; 'there is a time *early* that is earlier than t at which the state was different to which it is at t. The state does not change at any time from *early* up to (but not including) t.'

The function *disconts* returns the set of all discontinuities in an activity.

$$\begin{aligned} &disconts : Activity \to \mathcal{P}(\mathsf{T}) \\ &disconts(\alpha) \mathrel{\hat{=}} \{t : \mathsf{T} \mid discont(\alpha, t)\} \end{aligned} \tag{6.62}$$

Updates

We are considering machines that produce state changes and they will take a non-zero amount of time to do so. We also have to consider null state changes, where something happens — a computation performs its read or write phase but this does not effect the state. We therefore consider 'updates' which are pairs of times with at most one discontinuity between them.

An update function u maps the ordinal position of read and write phases to updates. Hence

the set U is all the update functions.

$$U \triangleq \mathbb{N}_1 \rightarrow (\mathsf{T} \times \mathsf{T}) \tag{6.63}$$

Again we split this one function into two derivative functions u_s and u_e which return the start and end times respectively of a read or write phase.

$$\begin{array}{l} \forall u : U \bullet \\ \quad u_s = \{(n \mapsto t) \mid \exists t' : \mathsf{T} \bullet (n \mapsto (t, t')) \in u\} \wedge \\ \quad u_e = \{(n \mapsto t') \mid \exists t : \mathsf{T} \bullet (n \mapsto (t, t')) \in u\} \end{array} \tag{6.64}$$

In words; 'for all functions u there are two related functions $u_s : \mathbb{N}_1 \rightarrow \mathsf{T}$ and $u_e : \mathbb{N}_1 \rightarrow \mathsf{T}$ which return the start and end times of the updates respectively.' Hence if a read/write phase occurs nth in the read/write ordering then the update caused by that phase starts at time $u_s(n)$ and ends at time $u_e(n)$.

The predicate *updates* takes an update function and an activity and holds true if the function has required properties with respect to the activity. *i.e.* it captures all the discontinuities in the activity as occurring in at most one update, all its updates occur in non-zero time and it maintains the ordering of the read/write ordering.

$$\begin{array}{l} \mathit{updates} : U \times \mathit{Activity} \rightarrow \mathbb{B} \\ \mathit{updates}(u, \alpha) \triangleq \\ \quad 1 \quad \mathit{contig}(\mathbf{dom}\ u) \wedge \\ \qquad \forall n : \mathbb{N}_1 \bullet n \in \mathbf{dom}\ u \Rightarrow \\ \quad 2 \quad |\{t : \mathsf{T} \mid t \in \mathit{disconts}(\alpha) \wedge u_s(n) \leq t < u_e(n)\}| = 1 \wedge \\ \quad 3 \qquad \exists \delta : \mathsf{T} \bullet \delta > 0 \wedge u_s(n) + \delta \leq u_e(n) \wedge \\ \quad 4 \qquad n + 1 \in \mathbf{dom}\ u \Rightarrow u_e(n) \leq u_s(n+1) \end{array} \tag{6.65}$$

In words; 'each of the numbers in the left column refer to the enumerated points below. For a function to be a correct update function with respect to an activity it must...

1. have a contiguous domain.

2. capture every discontinuity in the activity in an update and allow at most one discontinuity per update. *i.e.* the set of discontinuities that occur in each update is singleton,

3. result in each update taking some minimum non-zero time. *i.e.* there is some non-zero

time δ which no update is quicker than. Note that this property ensures that we do not have problems with Zeno's paradox.

4. have no overlapping updates. *i.e.* for each update $u(n)$ if there is an update after it then $u(n+1)$ must not start earlier than $u(n)$ finishes.'

6.7.4 Correctness between computation sequences and activities

Now we can put the interleaving and update functions together to define whether a sequence of computations is projected correctly onto an activity.

Recall that the predicate *do* takes a computation and four state/time pairs and holds true if those four state/times are a correct performance of that computation. If the computation c occurs nth in a computation sequence and we have functions i and u then the four state/time pairs from an activity α relating to that computation are as follows...

- the start state/time is $\alpha((u_s(i_r(n))))$. (Recall the use of double brackets to denote the state and time at a given time in an activity — formula 6.10.) That is the i_r function applied to n to get the ordinal position of the read phase of c and u_s applied to that ordinal position to get the time of the start of the update caused by the read phase.

- the next state/time is $\alpha((u_e(i_r(n))))$. That is the i_r function applied to n to get the ordinal position of the read phase of c and u_e applied to that ordinal position to get the time of the end of the update caused by the read phase.

- the penultimate state/time is $\alpha((u_s(i_w(n))))$. That is the i_w function applied to n to get the ordinal position of the write phase of c and u_s applied to that ordinal position to get the time of the start of the update caused by the write phase.

- the final state/time is $\alpha((u_e(i_w(n))))$. That is the i_w function applied to n to get the ordinal position of the write phase of c and u_e applied to that ordinal position to get the time of the end of the update caused by the write phase.

So the nth computation c is projected correctly onto the activity α by the functions i and u if...

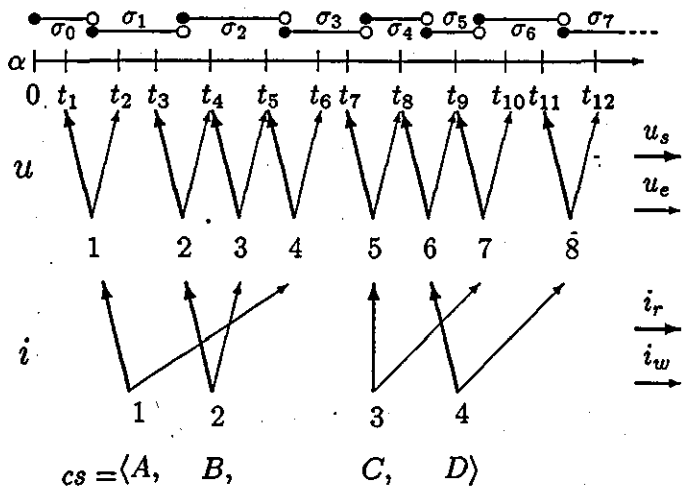


Figure 6-7: The computation sequence cs projected onto the activity α

$$do[\![c]\!](\alpha((u_s(i_r(n)))), \alpha((u_e(i_r(n)))), \alpha((u_s(i_w(n)))), \alpha((u_e(i_w(n)))))$$

The predicate *correct* returns true if all the computations in a computation sequence are correctly projected onto an activity.

$$\begin{aligned} &correct: I \times U \times C^{\Omega} \times Activity \rightarrow \mathbb{B} \\ &correct(i, u, cs, \alpha) \triangleq \forall n: \mathbb{N}_1 \bullet n \in \mathbf{dom}\ \alpha \Rightarrow \\ &\qquad do[\![cs(n)]\!](\alpha((u_s(i_r(n)))), \alpha((u_e(i_r(n)))), \\ &\qquad\qquad \alpha((u_s(i_w(n)))), \alpha((u_e(i_w(n))))) \end{aligned} \tag{6.66}$$

In words; 'for every n which is the ordinal position of a computation in cs the nth computation is projected correctly by i and u onto the activity α.'

To illustrate this predicate, consider figure 6-7 which shows the possible projection of the computation sequence cs onto the activity α. (This is effectively figure 6-5 in more detail.)

6.7.5 Obligatory computations

Before we can derive the behaviours of a specification there are two other matters we must deal with — obligation and fairness.

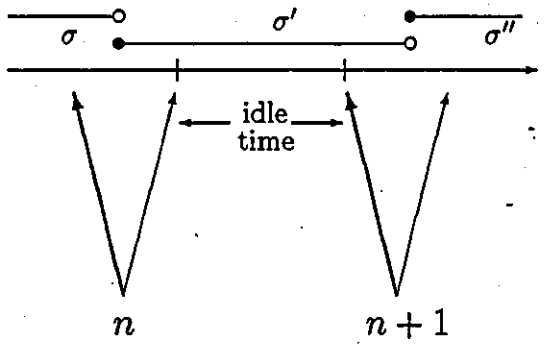


Figure 6-8: Idle time shown pictorially

Obligation asserts that the system cannot be idle while there are obligated computations enabled. In the definition of the update function we allowed for idle times in the activity. For a given n the start of the update $u(n+1)$ does not have to occur immediately after the end of $u(n)$, this period between two consecutive updates is known as 'idle time' (see figure 6-8). Returning to figure 6-7 there are idle times between t_2 and t_3, between t_6 and t_7 and between t_{10} and t_{11}. There can be no state changes during idle time because, by definition, all the discontinuities lie within updates.

The predicate *oblig* takes an update function, an activity and a set of computations and returns true if none of those computations are enabled during idle time in the activity.

$$\begin{aligned} &oblig : U \times Activity \times \mathcal{F}(C) \rightarrow \mathbb{B} \\ &oblig(u, \alpha, obl) \triangleq \forall t : \mathsf{T} \bullet \exists c : C \bullet c \in obl \wedge \alpha((t)) \in enabled[\![c]\!] \Rightarrow \\ &\qquad\qquad \exists n : \mathbb{N}_1 \bullet u_s(n) \leq t < u_e(n) \end{aligned} \tag{6.67}$$

In words; 'for all times t, if there is a computation from *obl* which is enabled at t then there must be an update $u(n)$ which t occurs in.'

6.7.6 Fairness

The notion of fairness we capture is that of 'strong fairness'. If an obligated computation is enabled infinitely often then it must occur infinitely often. For something to occur 'infinitely often' does *not* mean that it occurs all the time, but that at all times there is a future time when it occurs. We can express fairness in slogan terms using temporal logic operators as follows...

$$\Box\Diamond(A \text{ is enabled}) \Rightarrow \Box\Diamond(A \text{ is done})$$

To capture fairness we need reference to both an activity to define whether a computation is enabled infinitely often and a computation sequence to define whether a computation occurs infinitely often. The predicate *fair* captures fairness.

$$\begin{array}{l} fair : C^{\Omega} \times Activity \times \mathcal{F}(C) \rightarrow \mathbb{B} \\ fair(cs, \alpha, obl) \triangleq \\ \quad \forall c : C \bullet c \in obl \Rightarrow \\ \qquad (\forall t : \mathsf{T} \bullet \exists t' : \mathsf{T} \bullet t' \geq t \wedge \alpha((t')) \in enabled(c)) \Rightarrow \\ \qquad (\forall n : \mathbb{N}_1 \bullet \exists n' : \mathbb{N}_1 \bullet n' \geq n \wedge cs(n') = c) \end{array} \tag{6.68}$$

In words; 'for all computations c that are in *obl* if c is enabled infinitely often (for every time t there is a later t' at which c is enabled) in α then it occurs infinitely often in cs (for every ordinal position n there is a later position n' at which c occurs).'

6.7.7 The behaviour described by specifications

Now we can start putting things together. The predicate *perform* brings *correct*, *oblig* and *fair* together in a single predicate and also adds the condition that an activity must start in a state described by the initial property in a specification.

$$\begin{array}{l} perform : RSSL \times Activity \times C^{\Omega} \rightarrow \mathbb{B} \\ perform((init, obl, opt), \alpha, cs) \triangleq \\ \qquad 1.\ \ \exists i : I \bullet \exists u : U \bullet interleaving(i) \wedge updates(u, \alpha) \wedge \\ \qquad 2.\ \ (\mathbf{ran}\ i_r \cup \mathbf{ran}\ i_w) = \mathbf{dom}\ u \wedge \mathbf{dom}\ cs = \mathbf{dom}\ i \wedge \\ \qquad 3.\ \ prop[\![init]\!]\alpha((0)) \wedge \\ \qquad 4.\ \ cs \in (obl \cup opt)^{\Omega} \wedge correct(i, u, cs, \alpha) \wedge \\ \qquad 5.\ \ oblig(u, \alpha, obl) \wedge \\ \qquad 6.\ \ fair(cs, \alpha, obl) \end{array} \tag{6.69}$$

In words; 'given a specification, an activity and a computation sequence then *perform* holds true if...

1. there exist interleaving and update functions such that the interleaving function is correct and the update function is correct with respect to the activity,

2. the computation sequence and the two functions 'fit together' correctly. *i.e.* the domain of the sequence is equal to the domain of the interleaving function and the union of the ranges of the interleaving function is equal to the domain of the updates function,

3. the state at time zero in the activity is described by the initial property in the specification,

4. all the computations in the computation sequence come from computations described in the specification and the computation sequence is correctly projected onto the activity,

5. the obligation condition holds, and

6. the fairness condition holds.'

Using this we can describe the set of activities that are legal for a specification.

$$\begin{array}{l} s: RSSL \to Behaviour \\ s(spec) \triangleq \{\alpha: Activity \mid \exists cs: C^{\Omega} \bullet perform(spec, \alpha, cs)\} \end{array} \tag{6.70}$$

In words; 's returns the set of all activities α for which there exists a computation sequence cs which can correctly be applied to the predicate *perform*.'

This concludes the definition of the semantics for RSSL specifications, however we define one other useful function on RSSL specifications. The function *seqBeh* (standing for 'sequence behaviour') is similar to s, but returns the set of all computation sequences which are correct for a specification.

$$\begin{array}{l} seqBeh: RSSL \to \mathcal{P}(C^{\Omega}) \\ seqBeh(spec) \triangleq \{cs: C^{\Omega} \mid \exists \alpha: Activity \bullet perform(spec, \alpha, cs)\} \end{array} \tag{6.71}$$

In words; '*seqBeh* returns the set of all computation sequences cs for which there exists an activity α which can correctly be applied to the predicate *perform*.'

6.8 Refining specifications

If we crudely think that a refinement 'throws away' parts of the behaviour space then such an approach must maintain safety — if specification ψ is safe and we refine it to specification ϕ such that the behaviour of ϕ is a sub-set of the behaviour of ψ, then safety must be maintained.

All the activities in the behaviour of ψ are safe and the behaviour of ϕ is a sub-set of that of ψ so all the activities in behaviour of ϕ must be safe.

However 'throwing away' activities arbitrarily in a refinement does not guarantee liveness. In fact we could just throw away *all* the activities, so getting a system that is safe, because it does not do anything, but could hardly be described as live.

Traditionally refinement is a case of adding detail to a specification such that the behaviour space is constrained. This approach is usually based on specifications where liveness is implicit. However RSSL specifications do not guarantee liveness and therefore we need to show that a refinement maintains liveness.

Given a series of specifications that we propose is a refinement process as follows...

$$spec_1, spec_2, \ldots, spec_n$$

...where $spec_1$ is the most abstract specification and $spec_n$ the most concrete. We can theoretically define a safety property and a liveness property for each specification such that...

$$spec_i \Leftrightarrow live_i \wedge safe_i$$

...for each i such that $1 \leq i \leq n$. Specification $spec_{i+1}$ is a correct refinement of specification $spec_i$ (denoted $spec_i \rhd spec_{i+1}$) iff...

$$safe_{i+1} \Rightarrow safe_i \wedge live_i \Rightarrow live_{i+1}$$

In words; 'the refinement constrains the behaviour of the specification without losing liveness.'

Theoretically this is fine, but in practice it is unlikely that we would be able to divide an arbitrary specification into safety and liveness components like this. So we sketch an approach that demonstrates that safety and liveness properties are upheld by given specifications.

Assume we have requirements defined as follows...

$$req \hat{=} \Box(\phi) \wedge \Box(\psi \Rightarrow \Diamond \varphi) \tag{6.72}$$

...where $\Box(\phi)$ is the safety condition and $\Box(\psi \Rightarrow \Diamond \varphi)$ is the liveness condition. (ψ is an environment request and φ is a kernel response.)

Furthermore we have two RSSL specifications *spec* and *spec'*. The following two sections sketch out how we might go about showing that *spec* and *spec'* are consistent with the requirements and furthermore that *spec'* is a refinement of *spec*.

6.8.1 Safety

Proving safety for *spec* is a case of showing that all activities in its behaviour contain only safe state/times. *i.e.* ones consistent with ϕ.

$$\forall \alpha : Activity \bullet \alpha \in s[\![spec]\!] \Rightarrow \forall t : \mathsf{T} \bullet prop[\![\phi]\!]\alpha((t)) \tag{6.73}$$

In words; 'for each activity α in *spec*'s behaviour, for every time t the state/time pair at time t in α is safe.'

For *spec'* to be a refinement of *spec* then the behaviour of *spec* should contain no more activities than *spec*.

$$s[\![spec']\!] \subseteq s[\![spec]\!] \tag{6.74}$$

6.8.2 Liveness

There are two issues to deal with in liveness — firstly that the system always eventually gets to 'live' states and secondly how much the system deviates getting to those live states.

The liveness property $\Box(\psi \Rightarrow \Diamond \varphi)$ states that if a request ψ is made then there must eventually be a response φ. In other words, for all activities that are legal for a specification if at any time in that activity there is a state consistent with ψ then *all* legal extensions of that activity must contain a state that is consistent with φ.

Formally...

$$\begin{array}{l}\forall \alpha : Activity \bullet \alpha \in s[\![spec]\!] \Rightarrow \\ \forall t : \mathsf{T} \bullet prop[\![\psi]\!]\alpha((t)) \Rightarrow \\ \forall \alpha' : Activity \bullet \left(\begin{array}{c} \alpha' \in s[\![spec]\!] \wedge \\ \forall early : \mathsf{T} \bullet early \leq t \Rightarrow \alpha(early) = \alpha'(early) \end{array} \right) \Rightarrow \\ \qquad\qquad \exists t' : \mathsf{T} \bullet prop[\![\varphi]\!]\alpha'((t')) \end{array} \tag{6.75}$$

In words; 'for all activities α that are legal for the specification if there is a time t at which the request ψ is made then all activities α' which are extensions of α must show the response φ. By 'extension' we mean an activity that is legal for the specification and is the same as α up to time t'.'

This guarantees liveness for specification *spec*. More subtly we want to be able to capture that after a request is made the system always gets closer to the response, either until the system gets to the response or until another request is made. Assume we have a measurement scheme M on states and times.

$$M : \Sigma \times \mathsf{T} \rightarrow \mathbf{N} \tag{6.76}$$

...such that a state/time that is a response φ measures 0 and the highest measurement in the scheme is for state/times that are requests ψ. M therefore measures 'how far away' from the response state the system is. Formally...

$$\begin{array}{l}\forall (\sigma, t) : \Sigma \times \mathsf{T} \bullet prop[\![\varphi]\!](\sigma, t) \Rightarrow M((\sigma, t)) = 0 \wedge \\ \qquad\qquad prop[\![\psi]\!](\sigma, t) \Rightarrow M((\sigma, t)) = max(\mathbf{ran}\ M) \end{array} \tag{6.77}$$

We cannot sensibly assert that the value of M in an activity always decreases after requests, because that would lead us into problems similar to the stuttering problem, so we assert that the value of M must never increase (unless a request is made).

$$
\begin{aligned}
&\forall \alpha : Activity \bullet \alpha \in s[\![spec]\!] \Rightarrow \\
&\forall t : \mathsf{T} \bullet t \notin reqs \Rightarrow \\
&\qquad M(\alpha((t))) \leq M(\alpha((nearest(reqs, t)))) \\
&\textbf{where} \\
&reqs = \{t : \mathsf{T} \mid prop[\![\psi]\!]\alpha((t))\}
\end{aligned}
\tag{6.78}
$$

$$
\begin{aligned}
&nearest : \mathcal{P}(\mathsf{T}) \times \mathsf{T} \rightarrow \mathsf{T} \\
&nearest(ts, t) \hat{=} \infty \text{ if } t \geq max(ts) \\
&\qquad\qquad min(\{t' : \mathsf{T} \mid t' \in ts \wedge t' > t\}) \text{ otherwise}
\end{aligned}
\tag{6.79}
$$

In words; 'for all legal activities α there is a set of times *reqs* which is all the times at which requests are made. For any time t which is not in *reqs* all the times between t and the next request the value of M must not be any greater than it is at t. The function *nearest* returns the value from a given set of times that is closest but greater than a given time.'

The value of M may remain constant and the system never get anywhere according to equation 6.78, but by conjoining it with equation 6.75 we assure ourselves that a response is eventually reached.

We suggest that we can capture the refinement of liveness by defining different measurement schemes for different specifications. The most general measurement scheme is as follows...

$$
\begin{aligned}
\forall (\sigma, t) : \Sigma \times \mathsf{T} \bullet prop[\![\varphi]\!](\sigma, t) \Rightarrow M((\sigma, t)) = 0 \wedge \\
\neg prop[\![\psi]\!](\sigma, t) \Rightarrow M((\sigma, t)) = 1
\end{aligned}
$$

All state/times measure 1 unless they are a response, so the system can go through any sequence of state/times to get to a response.

We can then define other measurement schemes that progressively limit the sequences the system can go through to get to a response whilst still assuring that the system does get to a response eventually.

If *spec* is live based on a measurement scheme M and *spec'* is live based on M' then *spec'* is a refinement of *spec* if...

$$\forall(\sigma,t):\Sigma\times\mathsf{T}\bullet M((\sigma,t))\leq M'((\sigma,t)) \tag{6.80}$$

...*i.e.* each state/time in the more concrete specification is measured at least as far away from the response as they are in the more abstract. In other words the refined system can only deviate in getting from the request to the response as much as the abstract system can. A strict refinement will deviate less.

So for a specification *spec* to be a refinement of requirements *req*, denoted $req \rhd spec$, then formulae 6.73 and 6.75 must hold. For a specification *spec* to be a refinement of specification *spec'*, denoted $spec \rhd spec'$ then formulae 6.74 and 6.80 must hold.

6.8.3 Words of caution

This is only a sketch of how we may go about refining RSSL specifications. Actually proving these relationships would be very arduous. Proving safety is fairly straight forward. Most traditional specification languages (such as VDM and CSP) have liveness implicit in them, so any refinement preserves liveness by default. There is plenty of work showing how to refine such specifications, but in reality this is only a refinement of safety.

There is little work that we can tap into to show how we can refine liveness properties. There is work in progress being done by TLA researchers to capture liveness refinement but it is not as yet complete or published. Peter Ladkin said in a personal comment that refinement of liveness properties is the next big challenge for temporal logic specifications.

6.9 Reactions

Recall from the previous chapter (section 5.6.4) we described a reactive system as being a set of optional reactions, where a reaction is a pair consisting of an environment computation (known as the invocation) and a kernel computation (known as the response).

A reactive system specified in this way is simply a syntactic sugaring of a specification expressed as an initial predicate, a set of obligatory computations and a set of optional computations. Once again we are simply describing differences in style, rather than any fundamental substance.

Given a specification in the reaction style with just one reaction...

$$\Psi \mathrel{\hat{=}} init \wedge \Box\langle e \fatsemi k \rangle \tag{6.81}$$

...and the two computations e and k are defined as follows...

$$\begin{aligned} e \mathrel{\hat{=}} & \textbf{input} : eX \\ & \textbf{output} : eY \\ & \textbf{enabled\_by} : eE \\ & \textbf{side-effect} : eS \\ & \textbf{outcome} : eO \end{aligned} \tag{6.82}$$

$$\begin{aligned} k \mathrel{\hat{=}} & \textbf{input} : kX \\ & \textbf{output} : kY \\ & \textbf{invoked\_by} : e \\ & \textbf{enabled\_by} : kE \\ & \textbf{side-effect} : kS \\ & \textbf{outcome} : kO \end{aligned} \tag{6.83}$$

...then we say that k is invoked by e. These computations 'hide' a count $eCount : \mathbb{N}$ which is incremented every time the invocation occurs. The response is enabled whenever $eCount$ is greater than zero and its side-effect decrements $eCount$.

So e is a syntactic sugar for...

$$\begin{aligned} eSugar \mathrel{\hat{=}} & \textbf{input} : eX, eCount \\ & \textbf{output} : eY \\ & \textbf{enabled\_by} : eE \\ & \textbf{side-effect} : eS \wedge eCount^n = eCount + 1 \\ & \textbf{outcome} : eO \end{aligned} \tag{6.84}$$

...and k is a sugar for...

$$\begin{aligned} kSugar \mathrel{\hat{=}} & \textbf{input} : kX, eCount \\ & \textbf{output} : kY \\ & \textbf{enabled\_by} : kE \wedge eCount > 0 \\ & \textbf{side-effect} : kS \wedge eCount^n = eCount - 1 \\ & \textbf{outcome} : kO \end{aligned} \tag{6.85}$$

Overall the reaction style specification Ψ is a sugar for the following specification...

$$\Psi_{sugar} \triangleq init \wedge eCount = 0 \wedge \Box[kSugar] \wedge \Box\langle eSugar\rangle \tag{6.86}$$

The reaction style specification we have introduced is (once again) a technique for specifying 'typical' systems, which hopefully does not overly restrict us from specifying atypical systems, but makes us think about what we are doing if we do.

6.9.1 A specialisation of RSSL based purely on reactions

A reaction is a pair of computations.

$$React \triangleq (C \times C) \tag{6.87}$$

...where if $(e, k) : React$ then e is the environment computation and k the kernel computation.

RSSLreact is the set of all RSSL specifications based purely on reactions, *i.e.* where every environment computation is paired with a kernel computation.

$$RSSLreact \triangleq P \times \mathcal{F}(React) \tag{6.88}$$

In words; 'an RSSL specification based on reactions is an initial predicate and a finite set of optional reactions.'

We can convert any *RSSLreact* to an *RSSL* specification as follows...

$$\begin{array}{l} conv : RSSLreact \rightarrow RSSL \\ conv((init, reacts)) \triangleq (init, \{k : C \mid \exists e : C \bullet (e, k) \in reacts\}, \\ \qquad\qquad\qquad\qquad \{e : C \mid \exists e : C \bullet (e, k) \in reacts\}) \end{array} \tag{6.89}$$

In words; 'an *RSSL* specification is converted from a *RSSLreact* specification by separating the environment and kernel computations out from each reaction.'

Note that it is possible to turn every *RSSLreact* specification into an *RSSL* specification, but not *vice versa*. Specifications in the reaction style are a specialisation of *RSSL* specifications.

6.9.2 Semantics for specifications in the reaction style

Determining the behaviour for a specification the reaction style is simply a case of *conv*erting it into an RSSL specification and applying the semantic function s.

However a useful way of thinking about the behaviour of a reaction style specification is in terms of sequences of reactions that can be legally generated. We define two functions...

- $reactBeh^{\infty}$ which returns the set of all possibly infinite sequences of reactions which are legal, and
- $reactBeh$ which returns the set of all finite sequences of reactions which are legal.

In the next two chapters we shall discuss how we can describe the probabilistic behaviour of systems by attaching probability measurements to reactions in a specification. The probability of a sequence of reactions occurring is the product of the all the probabilities of individual reactions occurring. If a sequence is infinite then such a product cannot be calculated, so we define $reactBeh$ to return only the legal finite sequences.

A reaction sequence is legal for a specification if the computations that make up the reactions can be interleaved onto a sequence of computations that is legal for the specification. The notion of interleaving is exactly the same as we used to map computation sequences onto read/write orderings previously, so we can reuse the function *interleaving*. Given an interleaving function i then if a reaction occurs nth in a reaction sequence and $i(n) = (x, y)$ then its environment computation should occur xth in a computation sequence and its kernel computation should occur yth.

Formally...

$$
\begin{array}{l}
reactBeh^{\infty} : RSSLreact \to \mathcal{P}(React^{\Omega}) \\
reactBeh^{\infty}(spec) \triangleq \\
\quad \{rs : React^{\Omega} \mid \exists cs : C^{\Omega} \bullet cs \in seqBeh(conv(spec)) \wedge \\
\qquad \exists i : I \bullet interleaving(i) \wedge \mathbf{dom}\ i = \mathbf{dom}\ rs \wedge \\
\qquad \forall n : \mathbb{N}_1 \bullet n \in \mathbf{dom}\ rs \Rightarrow \\
\qquad \exists x, y : \mathbb{N}_1 \bullet \exists (e, k) : React \bullet i(n) = (x, y) \wedge \\
\qquad\quad rs(n) = (e, k) \wedge cs(x) = e \wedge cs(y) = k\}
\end{array}
\tag{6.90}
$$

In words; 'the set of all reaction sequences is returned such that there exists a computation sequence cs that is legal for the specification and there exists an interleaving function i that maps

the reactions in the reaction sequence onto the computations on the computation sequence.'

The function *reactBeh* simply delimits the set returned by $reactBeh^{\infty}$ to finite sequences only.

$$\begin{aligned} &reactBeh: RSSLreact \rightarrow \mathcal{P}(React^*) \\ &reactBeh(spec) \triangleq \{rs: React^* \mid rs \in reactBeh^{\infty}(spec)\} \end{aligned} \tag{6.91}$$

In words; 'all the finite sequences of computations that are legal for the specification are returned.'

Chapter 7

Describing the use of a system with an Interactive System Specification Language (ISSL)

In this chapter we describe a 'framework' for specifying how a device might be used by its user population. Our framework is similar in concept to the Interaction Framework (IF) of Blandford *et al.* [17]. We start by describing the behaviour of a system and then we use our framework to consider various issues about how that behaviour might be influenced. IF takes the description of the interaction itself as its starting point, describes properties of the interaction and then provides 'hooks' for device and user models. It can then be proved that the interactions generated by the composition of the user and device models are consistent with the properties described by IF. Both our framework and IF deal with 'non-functional' requirements for a system.

Being a 'framework' however, the approach described in this chapter is more about providing a context for analysts to ask questions about use and usability issues, rather than providing firm answers to these questions. In the absence of an accepted body of HCI theory, frameworks which separate concerns allow the analyst to gain answers from heuristics and craft expertise if necessary, and to feed those answers into a sensible context. In this way we are proposing structures that provide bridges into experimental psychology in the same spirit as the templates work [102]. We realise that HCI findings cannot be completely formalised and fed into an

interactive design process. The next two chapters show structures that are as mathematical as is sensible and attempt to make the design decisions that are made within the structures explicit, inspectable and reusable, as required by good design practice.

7.1 The behaviour and the use of systems

The previous two chapters described an approach for describing and specifying reactive systems. The next two chapters describe how we can take this approach and specialise it for interactive systems.

RSSL (Reactive System Specification Language) describes the legal behaviour of a system in a discrete way — either an activity is legal or it is not. Such a description of a system is wholly adequate for modelling computerised machinery. Computerised machinery tends to be based on discrete mechanisms and so a language based on discrete mathematics is appropriate.

However the user side of an interactive system is less well modelled using discrete mathematics. When a system is modelled discretely then we have to allow user behaviour a high degree of non-determinism. It possible to determine (at least) two sources of non-determinism in a system description, namely...

Specification non-determinism which results from abstraction in a system description. At abstract levels there will be non-determinism in a description, which essentially states that the specifier does not care how things occur as long as the effect is as specified. Consider the non-deterministic operator in CSP [64] $P \sqcap Q$. This specifies a system which behaves like P or Q. It is very unlikely that an implemented device would actually behave like P or Q, making a genuinely non-deterministic choice between the two at run-time. Instead it is up the designer to decide which is more suitable according to some design criteria and implement a system that behaves like P or a system that behaves like Q. The refinement process should remove specification non-determinism until the behaviour of the device is (as good as) deterministic.

User choice non-determinism which results from the user being presented with a collection of actions to invoke and making a choice between them. In CSP terms this kind of non-determinism is denoted by the choice operator; $(x \rightarrow P \mid y \rightarrow Q)$. x and y are distinct

events which are offered to the environment to choose from. The device then behaves like P or Q depending on which event is chosen by the environment. It is certainly well beyond the scope of this thesis to discuss whether human behaviour shows genuine free will or whether if we are reductionist enough we can describe human behaviour in a mechanistic way. We simply assume that at any one time an interactive system offers a finite collection of options to a user and that a user makes a choice between them.

There are two extreme approaches to user behaviour; as a random choice between options or as a mechanistic deterministic choice. We opt for a sensible middle ground, where we can describe the probability of the choice a user makes.

In a software engineering context it is important to delimit the user's options to a finite set so that we can describe everything the user can possibly do with the device and we can specify a device that cannot be 'defeated' by the user — the user cannot do something for which the device response is undefined. However in the context of usability analysis and user interface design we put more structure on the space of user choice non-determinism, describing in a probabilistic way what is or is not likely to occur.

In summary, the previous two chapters concentrated on the 'behaviour space' of a system, behaviour being the set of legal activities for a system. The following two chapters look at the 'use' of a system, use (or usage) being a probabilistic distribution over the behaviour space of a system.

7.2 Usage requirements and interface specifications

The use of system is captured in two ways. Firstly the overall use of a system is looked at in order to capture how 'good' the use is. Secondly a lower level look is taken at the user interface in terms of what options the interface biases the user to choosing. The two are related so that we can assess what effect changing the user interface has on the use of a system — whether it makes the use 'better' or not.

The two approaches capture the requirements and specification of the use of a system. The requirements are stated in terms of how good we require the use to be. The specification is stated in terms of what effect the interface should have in order that the use of the system is

that described in the requirements.

7.2.1 What effect a user interface has on use

A user interface modifies the use of the system. A 'good' user interface is one that improves the use of the system (where an 'improvement' is a modification for the better). Note that a user interface only changes the use of the system, it does not alter the behaviour space of a system. The user interface is not the only entity in an interactive system that will alter the use. The user's intentions and motivations will play a considerable role too.

We shall suggest some degree of separation between the alteration of the use caused by the user and alteration caused by the interface (known as the 'user effect' and 'interface effect' respectively). Such a separation enables us to discuss interfaces in a way which is independent of their user population and *vice versa.* However, such a separation cannot be complete and will be contentious. We discuss these matters in more detail in the discussion section 7.7.2.

7.2.2 A word processor example

To illustrate the concepts introduced in this chapter we use a word processor example. We assume some requirements have been stated and the following abstract specification has been drawn up.

There are three possible reactions; one for editing text, one for formating text and one for disk operations. There are four objects in the state space; an input device, the text, the formatting and the contents of the hard disk. The kernel computations alter the latter three objects. The users alter the input device, but at this level of abstraction we are not concerned about how they do it — only that the state of the input device is in some way altered.

The system starts in a state where there is no text in memory. The user cannot invoke formatting or editing until there is some text to edit or format *i.e.* some text has been loaded. All this is shown in figure 7-1. The formatting is considered to be like a 'filter' over the text (see figure 7-2) hence the computation to alter the text may also alter the formatting. If we insert a character at the insertion point in figure 7-2 then the italic 'filter' must extend by one character. *save* and *load* are functions that describe the reading and writing of data to and from the hard drive.

$$WP \triangleq init \wedge \Box(edit \vee format \vee disk) \quad (7.1)$$

$$inDev{:}\ldots \quad (7.2)$$

$$text{:}\ldots \quad (7.3)$$

$$formatting{:}\ldots \quad (7.4)$$

$$disk{:}\ldots \quad (7.5)$$

$$init \triangleq text = \mathsf{Null} \quad (7.6)$$

$$edit \triangleq editI \fatsemi editR \quad (7.7)$$

$$format \triangleq formatI \fatsemi formatR \quad (7.8)$$

$$disk \triangleq diskI \fatsemi diskR \quad (7.9)$$

$editI \triangleq$
output : $inDev$
enabled_by : $text \neq \mathsf{Null}$
outcome : $inDev \neq inDev'$ (7.10)

$editR \triangleq$
output : $text, formatting$
invoked_by : $editI$
outcome : $text' \neq text$ (7.11)

$formatI \triangleq$
output : $inDev$
enabled_by : $text \neq \mathsf{Null}$
outcome : $inDev \neq inDev'$ (7.12)

$formatR \triangleq$
output : $formatting$
invoked_by : $formatI$
outcome :
$formatting' \neq formatting$ (7.13)

$diskI \triangleq$
output : $inDev$
outcome : $inDev \neq inDev'$ (7.14)

$diskR \triangleq$
output : $text, formatting, disk$
invoked_by : $diskI$
outcome :
$disk' = save(text, formatting, disk)$
$\vee$
$(text', formatting') = load(disk)$ (7.15)

Figure 7-1: A word processor specification

The 'specification non-determinism' (caused by abstraction on the kernel computations) is inherent in the responses. $text' \neq text$ is a highly non-deterministic statement and this non-determinism is reduced by the refinement process. The user choice non-determinism is inherent in the optional nature of the environment computations. This non-determinism is not (particularly) reduced by refinement. It is this non-determinism we look at when investigating the use of a system. Hence for most of this chapter we assume that software engineers are engaged in refining the kernel responses whilst we look in more detail at how the users control the invocations.

7.3 Describing usage requirements

We want to be able to capture how 'good' the use of a system is.

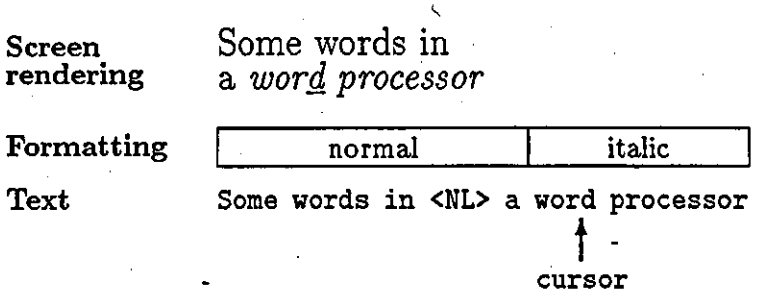


Figure 7-2: The formatting as a 'filter' over the text

7.3.1 Optimal behaviour

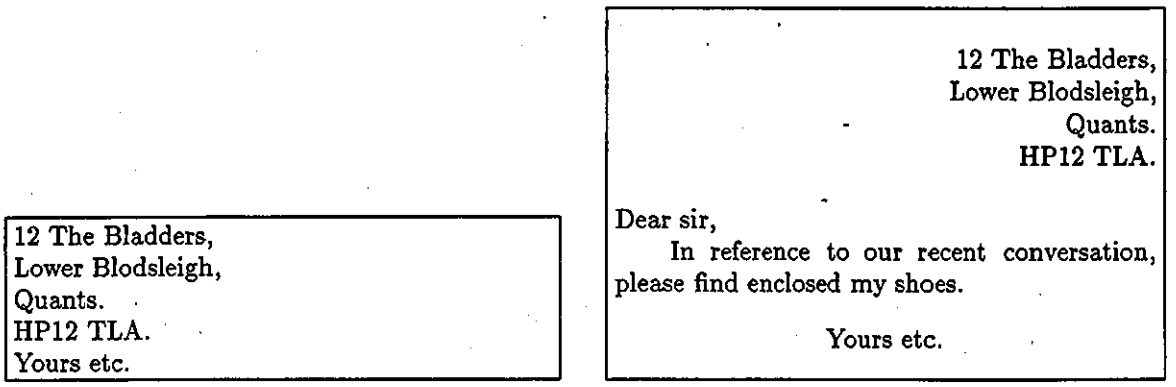
For every task there is an optimal way of achieving it with a given device. Blandford *et al.* [17] describe 'canonical trajectories' which are interaction paths that achieve a given goal in the most efficient manner possible. We describe such activities as 'optimal activities' (or optimal behaviour if we are talking about a collection of them). Conceptually, our notion of optimal activities is the same as these canonical trajectories.

By definition there is no better way of achieving a task than the optimal way. It is the behaviour that gets a task done most efficiently and with as little expending of resources as possible. A given task may have several different optimal activities. What is optimal is heavily dependent on the task; a user who has the task of editing a letter may be considered to be acting sub-optimally if she needs to access the help system to learn how to use the formatting tools. But if the task is 'edit a document whilst learning to use the formatting tools' then so long as the use of the help system is optimal the activity may be optimal.

A 'good' use of a system is one that is close to the optimal behaviour.

Assume a user employs the word processor specified in figure 7-1 to open a file with the contents shown in figure 7-3(a), edit and format that file so it looks like figure 7-3(b) and save the file. We also assume that we want the user to conform with good typesetting practice and edit the text before formatting it.

Assuming that *edit* inserts one character at a time and correctly formatting the address, the 'Dear sir,' line, the main body and the 'Yours etc.' line each requires one *format* then the optimal behaviour of the task is...



(a) The starting file

(b) The required file

Figure 7-3: Sample file contents for a word processor

$$WPoptimal = disk\ edit^{83}\ format^{4}\ disk \tag{7.16}$$

...where *WPoptimal* is expressed in a regular grammar-like notation. *i.e.* the optimal behaviour is a *disk*, followed by 83 *edits*, followed by 4 *formats* followed by a *disk*.

Of course, *WPoptimal* is extremely task specific, but we can define more general optimal behaviour if we were considering the word processor in more general use. We may decide (based on heuristic knowledge or experimental evidence gathered from observing expert users working with word processors) that typically the user performs approximately 20 times as many edits as formats when preparing any document. Hence the general case for optimal behaviour with a word processor may be described as...

$$WPgenopt = disk\ edit^{n}\ format^{m}\ disk \bullet n \approx 20m \tag{7.17}$$

The '•' reads 'where', hence the general optimal behaviour for a word processor is a disk operations followed by a series of edits, followed by a series of formats, followed by a disk operation, where there are around 20 times as many edits as formats.

This formula shows that a task may have several different optimal activities associated with it. As a rule of thumb we suggest that the more general the task definition, the greater the set

of possible equally optimal activities to achieve it.

7.3.2 Measurement schemes

Once we have captured the optimal behaviour of a system then we can consider how 'far' a given activity is from the optimal. We need a scheme over activities to measure this distance from the optimal.

We would like for HCI as a field to be in the position where we could propose measurement schemes and validate them against a body of inspectable HCI theory. Without such theory we can propose small scale schemes that describe user performance or we can propose larger, more general schemes that are necessarily contentious and open to question. We shall look at both approaches.

A measurement scheme based on user performance

Given the task of producing the document in figure 7-3 we assert that the user cannot achieve that task in less than 89 invocations. Therefore an activity that successfully achieves the task with 89 invocations is optimal. The more invocations it takes to achieve the goal, the less optimal is the activity. We can capture a measure of optimality as a ratio of number of erroneous invocations to the number of invocations made. We can simply calculate the number of erroneous invocations by subtracting 89 from the total number of invocations made.

If a user typed 'Dear madam,' then pressed the delete key 6 times and continued with 'sir,' and the rest of the letter successfully then he would take 101 invocations to achieve the task. Hence his error ratio would be...

$$\frac{101 - 89}{101} = \frac{12}{101} \approx 0.119$$

The greater the number of errors the greater the error ratio. An error ratio of 0 is optimal. By this calculation an error ratio of 1 is impossible, but that value can be used to indicate that the user fails to complete the task altogether.

There are a few problems with this measurement scheme; the user may format the address first, then enter the text and format the rest of the letter in 89 invocations. This activity is

not optimal according to *WPoptimal*, but scores a error ratio of 0. Hence we are in a good position to argue that the measurement scheme is inadequate. The question is, are we more concerned about the user getting the task done or are we worried about the user conforming to good typesetting practice? In the former case then this simple error ratio measurement is adequate. In more complicated tasks, failure to adhere to good typesetting practice can lead to extremely sub-optimal behaviour and we would need to capture this sub-optimality.

Towards a 'theory of measurement for optimality'

When users perform a task using a computer they will pass through a finite number of interaction steps, from some starting state to some state signifying the completion of a task. In the word processing example, we would define a user task as that of editing a document.

Again, much of the discussion here is very similar to the discussion of 'canonical trajectories' [17]. In our example the task starts with the loading of a document, is followed by several edits and formats and finishes with the saving of the document. We assert that expert behaviour is characterised by the user choosing the most efficient way from the start state to the end state in his task. We call this most efficient path an 'optimal path'. Distance from the optimal is proportional to the deviation from this optimal path.

That said, it is not clear how we measure deviation. We can identify three types of deviation;

- detours — the user unnecessarily meanders in her interaction,
- loops — the user finds herself in a situation she has previously been in without consciously backtracking, and
- errors — the user gets to a point in the interaction, realises she is in error and then has to retrace her steps to a point where she believes herself to be back on the right path again.

It could of course easily be argued that loops and errors are just special instances of detours, but we make the distinction because we want to be able to distinguish between different types of deviation so we can make a decision about which deviations are more undesirable than others. For example, a detour is probably preferable to an error of the same size because the user is more likely to become frustrated with errors.

This way of looking at a measurement scheme would work well creating ordering on activities that are very different, but it is much more difficult to order activities that are only slightly different.

All these ideas are wide open to debate and there are lots of woolly areas even in this short discussion; for example what do we mean by the most 'efficient' path? Is it the quickest in terms of time, or the one that requires least input from the user, or the one that places least cognitive load on the user, or the one that gets the task done efficiently whilst still allowing the user to feel he is in control of the task (as suggested in [89]), and so on. Furthermore are two small errors worse than a big loop?

Also to take into account is how well the task is performed — do we allow the completion of the task to be signified by the production of a result that is *approximately* similar to the result required? In our example, if the user produces a document with 'conversation' spelt incorrectly very efficiently then is this better than a user who produces a letter identical to that required, but produces it inefficiently?

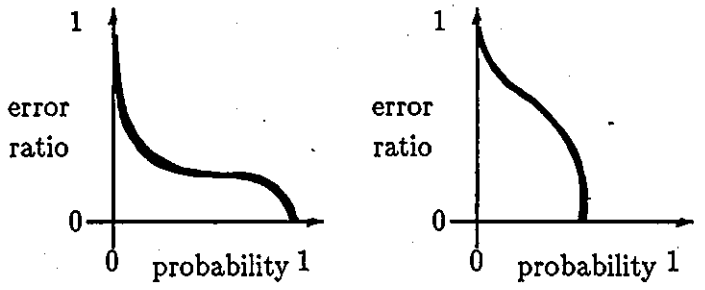
There needs to be much more work in HCI to be able to answer these questions in a way that is general to all (or even most) tasks. In the meantime we need to proceed taking the context of the task into account and relying on common sense and craft skill.

7.3.3 Usage distributions

Assume that we prototype a device based on the kernel specification in figure 7-1 and give it to n users to perform the task shown in figure 7-3. Each user attempts the task and we observe what they do. Using the error ratio scheme we measure how close to the optimal is what each user does.

Plotting the error ratio against the normalised frequency of users with that error ratio (the normalised frequency is a real number between 0 and 1; 0.5 representing 50% of the users and so on) we possibly would obtain graphs that look like those shown in figure 7-4. We call such graphs 'usage distributions'.

Note that the normalised user ratio is effectively the same as probability, a usage distribution can be thought of as mapping an error ratio (or whatever measurement scheme is used) to the probability of activities with that error ratio occurring.



(a) A fairly optimal distribution

(b) A less optimal distribution

Figure 7-4: Example usage distributions

Figure 7-4(a) shows more optimal use than that in figure 7-4(b) — more users show more optimal behaviour. The 'better' the system the greater the proportion of its usage distribution falls in the lower error ratios.

We are not aware of any empirical work that would allow us to predict the shape of usage distributions. We assume that the curves fall into regular distributions because common sense predicts they should. A usage distribution is a description of how effective users are at performing a task and therefore the distributions are unlikely to be purely random. If they are then we must fundamentally question HCI as a field. If the use of an interactive system *is* purely random then there is little point studying it and attempting to propose interfaces that modify the use for the better.

We consider usage distributions to be the 'requirements' for the use of an interactive system. They describe the use of a system in manner that is independent of the interface implementation. In the next section we consider the specification of uses — *i.e.* the description of systems that will give rise to the required uses as described by a usage distribution.

7.4 Specifying interactive systems

Consider the specification in figure 7-1. Once the user has started the device and performed a disk operation (initially *disk* is the only invocation that is enabled) she is offered a non-deterministic choice of what to do next. By repeatedly observing what choices the users make in this situation

we can assign probabilities to each of the invocations. In doing so we are moving the system model out of the realm of discrete mathematics which only describes a set of non-deterministic choices into the realm of probabilistic mathematics where we can describe what the user actually chooses to do in a more expressive way.

In any situation the total probability of what the user can do next is 1. In all likelihood these will not be static, context-free probabilities; what the user does next will be to a large degree determined by what they have already done and the visible state of the device.

What the user has done so far and the visible state of the device can be calculated from the sequences of reactions that leads up to any given choice being made. Therefore instead of attaching simple probabilities to each invocation we attach functions which take the interaction so far as an argument and returns the probability of an event occurring next.

We attach the following three probability functions to each of the invocations...

$$p_{edit} : React^* \rightarrow [0,1]$$
$$p_{format} : React^* \rightarrow [0,1]$$
$$p_{disk} : React^* \rightarrow [0,1]$$

...where after any reaction sequence the total probability returned by the functions is 1. Formally...

$$\forall rs : React^* \bullet p_{edit}(rs) + p_{format}(rs) + p_{disk}(rs) = 1$$

7.4.1 Interface and user effects

Having captured these probabilities the important question is what causes them; why is one choice more or less likely than another? There is no simple answer of course, but we argue that we can separately consider the user interface and the users' intentions, which we refer to as the 'interface effects' and 'user effects' respectively.

The interface effect is what is fundamental about an interface, what biasing effect it has on the use independent of the user population.

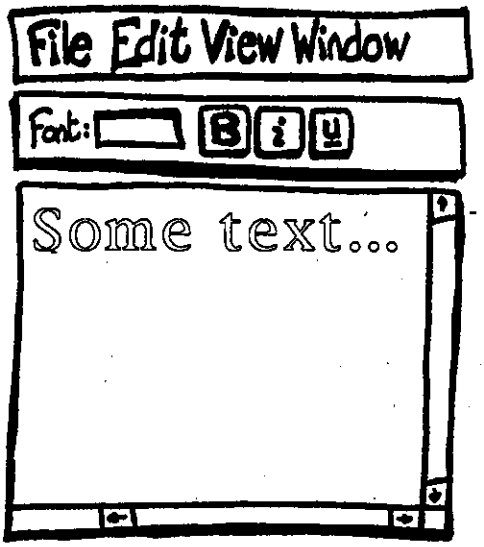


Figure 7-5: An example word processor interface

The user effect, conversely, captures what is fundamental about the user population; what they will try to do with the device no matter what interface they have to work with. User effect is therefore a blanket term covering all aspects of the user side of a system; their beliefs, motivations, intentions and so on.

An example interface effect

Consider a user interface to the word processor. The interface is shown in figure 7-5 and is based on the earlier versions of Word for Windows. We can present an estimation of the interface effect for this interface as follows.

Editing takes place in a prominent window and editing commands are simple and will generally have a one-to-one relationship to device commands; for instance the edit command 'insert(i)' is invoked by pressing the key marked 'i'. The interface therefore gives editing a very high probability; 0.95. Formatting is caused by pressing graphical buttons on a less prominent tool bar. This requires more effort; moving the mouse or a sequence of command keys. The interface probability of formatting is low; 0.04. The disk operations are in the menu system and therefore require more effort still and are therefore even less likely to be invoked; 0.01.

This is effectively a 'reverse engineering' of the interface. The use of probability measures

enables us to capture 'what' the interface does — the biasing it exerts on the user, rather than 'how' the interface causes this biasing. We want to be able to produce an 'interface specification' which describes in probabilistic terms what biasing the interface has on the use. We can then pass this interface specification to a human factors specialist who will 'know' what presentation issues and interface designs will produce this biasing.

This interface is 'static' — it does not alter during the interaction, but by using probability functions to describe the interface effect we can describe 'intelligent' interfaces that adapt themselves to the user. For example we could build an interface that prompts the user to perform disk operations if she has not done so for a long time. Hence we could define probabilistic functions that increase the probability of a disk operation occurring (and correspondingly decrease the probability of edits and formats) after a fixed number of invocations that do not include a disk operation. The degree of increase in the probability of a disk operation will be dependent on how the prompting is implemented. Presumably a dialogue box containing the text 'You may wish to perform a disk operation now' will increase the probability less than a dialogue box which contains 'Do a disk operation. Now.' and is accompanied by a warning sound. We are however, only really interested in the probabilities rather than the actual interface features that cause those probabilities.

User effects

The proposing of user effects is a possible point of contact in our approach to user modelling [25, 11, 118]. We would like to be able to employ a user modelling technique that could feed predictive results into our approach without having to resort to to expensive user tests to analyse every design decision.

To feed the results of a user modelling analysis into our approach we would need to 'translate' the results into a collection of probability functions. This translation is beyond the scope of this work but should figure very highly in the 'proposed further work'. Interfacing a good user modelling approach to our work will allow us to make valid design decisions earlier in the design lifecycle.

The user effect is much less likely to be static than the interface effect. What the user wants to do next will be heavily influenced by what they have done so far. We could model a fairly

competent user performing the simple letter editing task with a collection of functions that model the probability of the first invocation being a disk operation as being 1, then the probability of *edit* being very high until around 85 *edit*s have been performed, then the probability of *format* becomes high until 5 have been invoked and then the disk operation becomes very likely. We could then take this model and capture a less competent user as one where the difference between the probabilities is not so pronounced.

At the other extreme we can model the user population of a 'walk up and use' system as being simply random. In the case of our word processor the user effect shows no difference between each option; each invocation has the probability of 1/3. Note that this should *not* be the same as a model of novice users; novice users will have characteristics that are not just purely random. However a walk up and use system assumes an infinite user population and therefore ultimately the probabilities should 'average out' making all options equally likely.

Combining user and interface effects with probabilistic filters

We can think of the interface effects and user effects as 'filters' over the core system. 'Filtering' is a way of combining the interface and user effects so as make predictions about what the user will do next.

To illustrate this filtering consider two very simple user effects and the simple static interface effect described above. Firstly an extreme user who just wants to perform edits. Editing has a probability of 1, formatting and disk operations have a probability of 0. In this extreme case the interface has no effect; the user will only perform edits no matter what the interface tries to bias her to. See figure 7-6(a); the column u lists the user effect probabilities, the column i lists the interface effect probabilities and the column E lists the overall effect of the user effect filtered through the interface effect.

Another user (figure 7-6(b)) only wants to do formats or disk operations and is not bothered which. The overall effect is that edits have a probability of 0, formats have a probability of 0.8 and disk operations have a probability of 0.2. As the user does not care which of formats or disk operations she does then the biasing is due to the interface; the interface makes formats four times more likely than disk operations.

The filtering is calculated by first multiplying the probabilities of each pair of user and

	u	i	E
edit	1	0.95	1
format	0	0.04	0
disk	0	0.01	0

(a) A user that only wants to do edits

	u	i	E
edit	0	0.95	0
format	0.5	0.04	0.8
disk	0.5	0.01	0.2

(b) A user that only wants to do formats or disk operations

	u	i	E
$option_1$	u_1	i_1	$u_1 \times i_1 \times 1/T$
$option_2$	u_2	i_2	$u_2 \times i_2 \times 1/T$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$option_n$	u_n	i_n	$u_n \times i_n \times 1/T$

$$\text{where } T = \sum_{1 \leq j \leq n} (u_j \times i_j)$$

(c) A general approach to filters

Figure 7-6: Modelling probabilistic filters

interface effect. These multiplied probabilities are then normalised by multiplying each one by $1/T$ where T is the sum of all the multiplied probabilities. This general case is shown in figure 7-6(c). Note that it is possible to define filters that are inconsistent with each other such that each event is defined as being impossible by either the user or interface effect. Hence T is zero and $1/T$ has no meaning. Obviously a device that offers no choices that the user has any intention of doing is extremely problematic, so such inconsistent filters will be rarely observed and be avoided if specifying an interface.

The Interactive System Specification Language (ISSL)

An Interactive System Specification Language (ISSL) is based on reaction style RSSL, where user and interface effects are included in the invocation definitions. We include the probability functions for each invocation by adding extra clauses into their specification. The specification of the invocations in a word processor based on the simple interface and a 'walk up and use'

user is given in figure 7-7.

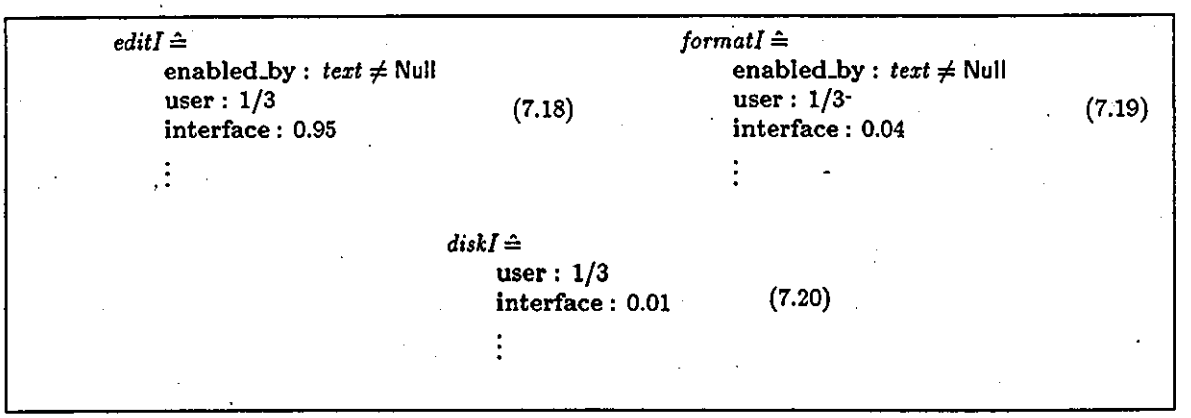


Figure 7-7: Adding probability measures to the invocations in a specification

If we need to attach probability functions rather than static probabilities to the invocations then we can name those functions, include the name of the function in the specification of the invocation and define the function underneath the specification. For example the specification of *diskI* for a walk up and use user and the 'intelligent' interface we described in section 7.4.1 is shown in figure 7-8.

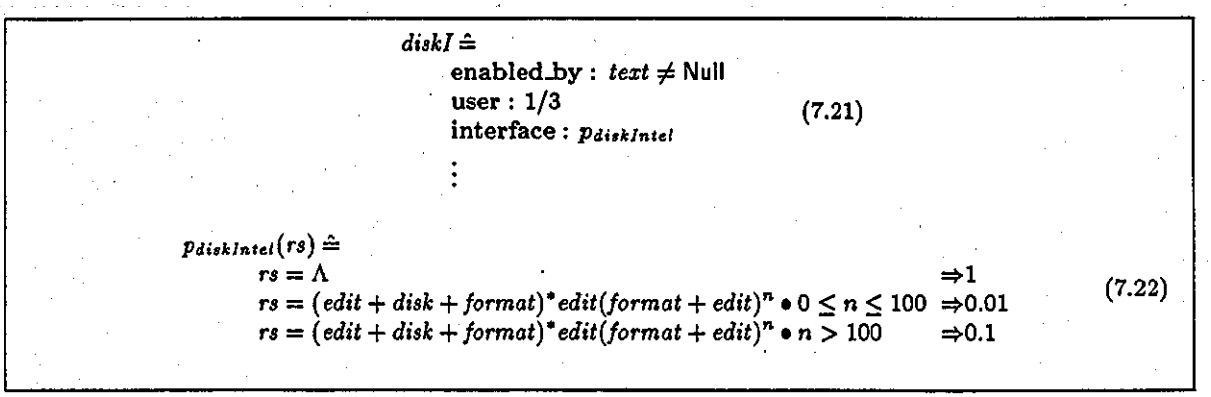


Figure 7-8: Defining probability functions for the invocation specifications

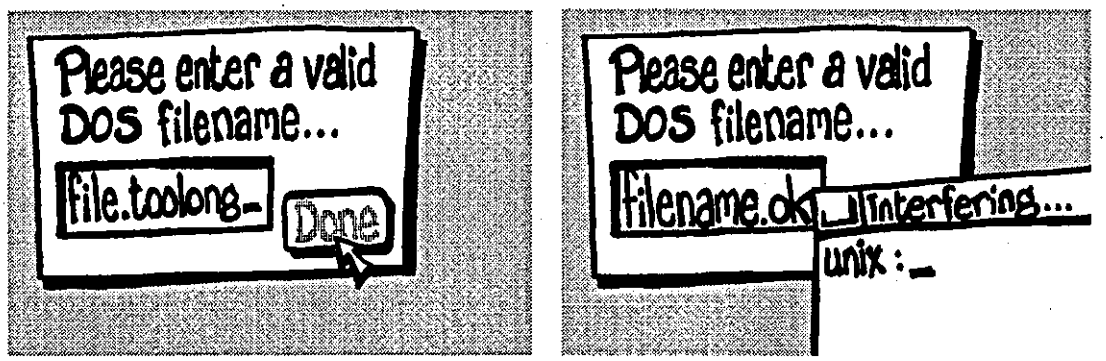
The interface effect for editing remains at a constant 0.95 until the user has performed 100 consecutive edits, then the interface reduces the probability of edits occurring. To capture this we define a probability function $p_{diskIntel}$ (standing for 'probability of *disk* with an intelligent interface').

In the notation for describing the probability function *rs* is the reaction sequence so far. The body of the definition lists various values for *rs* (using the regular grammar-like notation we used for describing optimal behaviour) and their corresponding probability values after the $\Rightarrow$ symbol. If nothing has happened yet ($rs = \Lambda$) then it is impossible to perform edits or formats, hence the probability of disk operations is one. The next line in the function definition describes all the activities for which there have not been any disk operations in (up to) the most recent 100 invocations. In this case the interface effect probability of a disk operation is 0.01. The last line describes activities for which there have not been any disk operations in the most recent 100 invocations and in this situation the probability of a disk operation increases to 0.1. (Note that we would have to define the probability functions for the edits and formats to decrease correspondingly once the probability of the disk operation increases.)

Impossible or illegal?

Note that if the 'enabled by' condition fails then the probability of the invocation must be zero, but not *vice versa*. Enabling conditions denote the legality of computations, the filter clause denotes the possibility of computations. An illegal computation is necessarily impossible, but an impossible computation is not necessarily illegal. (An 'illegal' computation is one for which the enabling condition has failed. An 'impossible' computation is one with a probability of zero.) Hence in figure 7-7 the enabling conditions in *editI* and *formatI* override the interface effect and set it to zero if the enabling condition fails.

Consider the example shown in figure 7.4.1. The user is offered a dialogue box and they should enter the name of a file to save in a text slot and then click on the 'Done' button. Until there is a valid file name entered in the slot the 'Done' button is greyed out and cannot be pressed; the invocation of pressing the button is illegal until a valid file name is entered. However imagine a system with overlapping windows such that the 'Done' button is covered by another window and the user cannot get to it without moving one of the windows. In this case even if a valid file name is entered in the slot then the invocation of pressing the 'Done' button will still be impossible. The enabling condition is about functionality and the interface and user effects are about the use.



(a) An illegal invocation

(b) An impossible invocation

Figure 7-9: The difference between an impossible and illegal invocation

Refinement of ISSL specifications

If we decompose a reaction into a collection of sub-reactions then we need to show that the total probability of the sub-reactions is equal to the probability of the reaction from which they were decomposed.

Assume the software engineers who are refining the reactions have split them into *insert*, *delete*, *moveCursor*, *copy* and *paste* where each one is composed of an invocation and response...

$$insert \mathrel{\hat{=}} insertI \fatsemi insertR$$

...and so on.

If p_{ins}, p_{del}, p_{mC}, p_{copy} and p_{paste} are the probability functions for *insertI*, *deleteI*, *moveCursorI*, *copyI* and *pasteI* respectively (calculated by filtering the user effect through the interface effect defined for each one) then the refinement of the probabilities is a case of showing that...

$$\forall rs : React^* \bullet (p_{ins}(rs) + p_{del}(rs) + p_{mC}(rs) + p_{copy}(rs) + p_{paste}(rs)) = p_{edit}(rs)$$

...(recall that p_{edit} is the probability function for *editI*).

7.5 Relating usage requirements and interface specifications

The previous two sections have described alternative ways of describing the use of interactive systems. Firstly in an 'overview' manner — attempting to capture what constitutes a 'good' use. Secondly, looking in more detail at the mechanics of the usage, what interface features and user intentions and motivations effect the use of the system.

In the next chapter we shall define semantics for both usage distributions and ISSL specifications. Hence we can formally assert whether a usage distribution and an ISSL specification are describing the same use, and hence are 'consistent' if they are.

A usage distribution determines the requirements of the use of a system. It is possible to define detailed graphs for the requirements, but more likely are statements such as 'the users have at least a 0.5 probability of performing their tasks with a error ratio less than 0.05' which we can translate into usage distributions.

The user effect constitutes the assumptions about the user population and the interface effect constitutes the specification of the interface. Given requirements for the use and a set of assumptions about the user population we can design an interface specification which filtered through the user effect produces the required use. In user interface terms the crucial question is what effect does a change in the probabilities attached to the interface specification of the interface have on the usage distributions? The goal of an interface designer is to 'tweak' the values of these probabilities in order to achieve a greater proportion of usages with a required low error ratio.

The specified interface effect is then passed to a human factors specialist who will know what interface features to implement in order to achieve the biasing required by the interface effect. One way of making our approach more practical would be to devise a collection of interface widgets in a toolkit where the biasing effect of each widget is known and documented in the toolkit.

7.6 A synthesis process for interactive systems

We now describe how interactive systems might be synthesized. The process contains two (broadly) concurrent streams; one dealing with the behaviour of the system and one dealing

with the use of the system.

The process 'starts'[1] with the definition of the functional requirements in terms of safety and liveness properties. A very abstract system can be specified which is a refinement of these requirements. This specification is expressed in terms of a set of reactions which the user invokes and it defines a space of legal behaviour. The set of reactions will be suggested by the result of a task analysis. Given a task, we wish to know what 'tools' will be useful in the achievement of that task. In our example we have implicitly assumed that the task of producing a document will be helped by the tools *edit*, *format* and *disk*. We can think of a reaction being a 'tool' for a user to employ in an interactive system.

At this point the process can split into its two streams. A more 'traditional' software engineering stream which deals with the functional refinement of the responses in the specification and an interface stream which deals with the invocations — how the tools are used by the user. A required use is proposed as are assumptions about the user population and an interface effect is specified. The output produced by the responses is also going to have an effect on how the user invokes reactions. Hence the software engineering stream feeds information about the output of the responses into the interface stream so that decisions can be made about how that output is presented to the user. The interface specification can then be passed to human factors specialists who know what actual interface features have the effect specified in probabilistic terms by the interface specification.

Each tool determines a task so we can apply a task analysis to each tool, which defines a more refined set of tools. The *edit* reaction might be decomposed into *insert*, *delete* and *moveCursor*. The composition of these three reactions should be a functional refinement of *edit* and the total probability of them occurring in any situation should be equal to the probability of *edit* occurring in that situation.

This refinement proceeds inductively until the responses can be translated into software and the invocations can be mapped onto low level device commands such as key presses and mouse movements. Ideally we would like to be able to 'design' the device commands, but often the hardware will be given so the design will have work 'towards' a collection of device commands

[1]This where the *formal* approach starts, there will be some body of semi-formal or completely informal requirements gathering before this.

that are available. However the decision of which invocations are mapped to which (collection of) device commands should be delayed as much as possible. Typically there will more invocations than there are device commands and so there will need to be a 'syntax layer' that translates sequences or combinations of device commands into invocations.

7.7 Discussion

The ideas presented in this chapter show many similarities to the Interaction Framework [10, 17]. However IF starts by describing interactions in a way that is abstract from the users and devices that cause those interactions, and then provide 'hooks' to device and user modelling so that interaction properties can be defined and then user and system models devised that when interacting with one another will fulfill those properties.

We start with user and device models expressed in an RSSL specification; these delimit what behaviour is legal. We then look at how such a behaviour is used with usage distributions and then define how that use is caused by an ISSL specification. Hence we build extra non-functional information into the functional description of a system whereas IF develops functionality from the non-functional requirements described for a system.

We have made several simplifying assumptions in this chapter. These points are discussed below.

7.7.1 Measurement schemes

We have illustrated our approach with a measurement scheme based on number of errors made. It could be argued that our approach is therefore more about improving performance than about improving usability. We take the stance that the point of making a device usable is to improve the user's performance of their given tasks. Therefore requiring a high level of performance from a system equates to requiring a high level of usability.

Other measurement schemes are possible. For example an analyst might decide that usability is more about how happy users are after having performed a task with a given device. The analyst could observe users performing a task and then apply a psychometric test to determine how happy they are afterwards. Presumably some activities make users happier than others and

the analyst could then specify interfaces that make those activities more likely.

We can change the measurements scheme, but this does not require that we change the framework in which the scheme sits.

7.7.2 Separation of user and interface effects

In an ideal world we envisage a function which would generate the probability of what happens next and would be parameterised by the user effect u and the interface effect i. By changing u, (*i.e.* by observing what different classes of users do with the system,) or by changing i (*i.e.* by giving different interfaces to the same users) we should theoretically be able to study why certain classes of user work well (or otherwise) with different types of interface.

However it would not be difficult to argue that it is impossible to change i without having an effect on u and *vice versa*. The interface presents the device to the users and colours the users' beliefs and perceptions about the device, so changing the interface will also change the users' beliefs. An interface is not just a simple channel of communication between users and the device, its role is more profound and subtle than that. This explains the worries expressed by some authors [108, 32] about drawing strict boundary lines between functionality and interface.

However, different users *do* make different uses of interfaces so there must be some level of separation to be made even if it is not the clean, simple separation we have suggested here. We suggest a move *towards* separation of user and interface effects is beneficial even if a complete separation is impossible.

7.7.3 Other implications of the use of the framework

Using our framework also allows us to capture interesting ideas about interactive systems we have not gone into detail with in the body of this chapter. Some of those ideas are catalogued below.

Interface construction as engineering

Our model of user-interface allows us to put the endeavour of interface building into an engineering context, where the necessity of building good interfaces is set against the cost of building the interfaces.

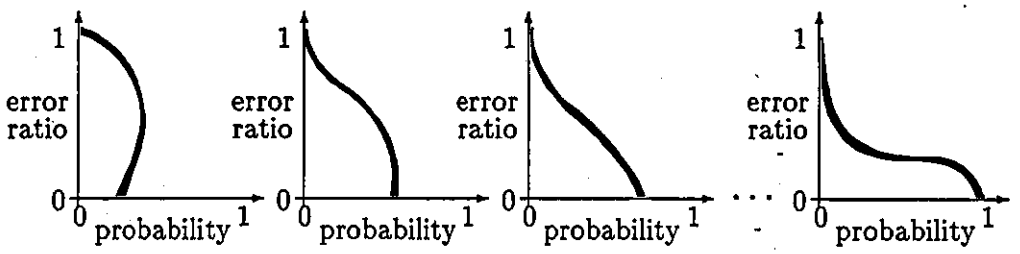


Figure 7-10: The effect of learning

We can get an idea of the comparative 'powers' of interfaces by looking at the amount of alteration they cause to usage distributions. The more alteration, the more powerful the interface. It is not unreasonable to assume that more powerful interfaces are going to be more expensive to produce.

We can therefore set the cost of producing the interface against the potential pay-off. Assuming the user population are employees of the client we are building a system for, an improved interface will presumably result in improved efficiency for the employees and an increase in productivity and profits for the client. This increase in profits can be set against the cost of building the interface. Another example is that of producing software systems for general use; 'user-friendliness' is a positive selling point and we can set the cost of producing a good user interface against the projected increase in sales due to the interface.

An important factor that can also be considered is necessity of user training. If an interface is needed that is prohibitively expensive then we could alter the other side of the equation by training the user population to act more like expert users, thereby reducing the needed power of the interface and the cost of building it. Of course training involves cost too, and the two should be traded off against each other.

Learnability

We can also express learnability with usage distributions. Assume we have users performing the same task n times, then we propose that we will get a series of distributions like that shown in figure 7-10. As the user learns to use the interface the usage distributions will become more biased to the optimal. We can capture this as the degree of change in the distribution with respect to the number of usages. We call this degree of change the 'learnability'.

We would be given a required learnability, the assumed learnability of the user population and we would calculate the necessary learnability to be built into the interface to satisfy the requirements.

Our framework contrasted to the 'usability properties approach'

We can contrast our approach to interactive systems to the 'usability properties approach' typified by [41, 61, 112]. We attempt to characterise behaviour and usage that result from a usable device and then design interface effects which will bring about this sort of behaviour and usage.

The usability properties approach considers various properties that are held (with varying levels of confidence) to have relevance to usability; for example, predictability, visibility and so on. Ultimately one would wish to have a set of properties the fulfillment of which guarantees usability. This approach therefore considers usability to be a composite of sub-properties whereas the approach reported here considers usability in a more 'holistic' way.

The two approaches are complementary. For example an analyst may make the assumption that users are likely to get easily lost, a fact that is expressed in a user effect. To prevent this the analyst may make the decision that a highly visible interface be implemented (based on the definition of visability given in [61]) and this notion of visibility can be defined in the interface effect. The usability property is then a method guiding the implementation of an interface that biases the users away from getting lost.

7.8 Conclusion

Our approach defines a formal framework for asking questions about usability and user interfaces; the answers to those questions then feed back to help the synthesis of an interactive system. Such a framework constitutes an attempt to move HCI work into a more rigorous, theoretical context. However our framework is *only* a framework and is therefore more about asking questions than providing answers. The crucial point is that we have a sensible and well defined context into which to feed those answers.

A framework also helps in proposing an agenda for further work. We suggest that in order to use our framework in a more predictive manner we need to pay more attention to user models,

or at least to how we interface the results of existing user models to our approach.

Chapter 8

A formal semantics for ISSL

In the previous chapter we introduced several concepts and ideas which we now give formal descriptions of. The chapter was about describing the use of systems, so firstly a formal description of use is given (section 8.1), followed by a definition of usage distributions and their semantics (section 8.2) and a definition of ISSL specifications and their semantics (section 8.3). This section also shows how we can compare uses described by usage distributions and ISSL specifications. Sections 8.4 and 8.5 give the formal definition for the other objects used in chapter 7, namely the regular grammar-like notation used for describing optimal behaviour and the notation for describing probability functions. We conclude with a discussion of the mathematical entities we have chosen to employ in this chapter (section 8.6).

8.1 A formal definition of use

The model of use we shall propose is based on sequences of reactions mapped to the probability of that sequence occurring. A use is therefore a probability distribution over sequences of reactions.

$$Use \mathrel{\hat{=}} React^* \rightarrow [0,1] \tag{8.1}$$

The total of all probabilities in a use should be one, hence...

$$\forall u : Use \bullet \sum_{rs : React^*} u(rs) = 1 \tag{8.2}$$

Note that a use is a distribution over all possible reaction sequences as it is a total function. Reaction sequences that are illegal for a given system are simply modelled as having a probability of zero.

8.1.1 Why such a simple model of use?

The definition of use we are working with is deliberately restrictively simple. A more general model may be a function from activities to probabilities, as follows...

$$Use \triangleq Activity \rightarrow [0, 1]$$

...and indeed it would be possible to construct the semantics we shall describe in this chapter based on this model, but it would be significantly more complicated and would obscure the main formal points of this chapter.

This chapter shows *one* way of dealing with models of use. We do not claim it to be the most general way, but we do claim it to be one of the most simple and clear. In section 8.6 we sketch out how we might go about describing use based on the activity model and look at the pitfalls in doing so. We list some of those pitfalls here to convince the reader of the benefits of the simple model of use.

- The set $React^*$ is countably large, whereas the set *Activity* is uncountably large (owing to its real time index). In an uncountably large space it is meaningless to assign probabilities to individual members of that space — they would all have zero (or at most infinitesimally small) probabilities. Hence we would have to 'approximate' activities into sets of similar activities in order to make the space countable. This is an extra layer of complexity not needed with $React^*$.

- Activities are infinitely long whereas sequences of reactions are by definition finite. The probability of a sequence of events occurring is calculated by the product of all the constituent events occurring. If the sequence is infinitely long then the probability of that sequence is likely to be zero.

- A specification can describe the same activity as being caused in several different ways —

a user may save a file by selecting the appropriate option in a menu system or by pressing F12. At a certain level of abstraction these will result in the same activity (*i.e.* the model of the hard-disk will be updated by according to the value of the currently open application) but this activity can be caused by (at least) two different invocation sequences. As we will calculate the probability of an activity from the probability of its constituent invocations then a given activity may have several probabilities associated with it. Again this is a level of complexity that obscures the main ideas. By considering only the sequences of reactions we do away with this extra level of complexity.

8.2 Usage distributions

Section 7.3.3 showed usage distributions as a function from error ratio to probability. However we then argued that error ratio is not necessarily the best measurement scheme to use to evaluate how well a user is using a device. We argued that it is not currently possible to propose a general measurement scheme to capture this 'goodness' of use and so we proposed that any sensible measurement scheme could be employed depending on the context of the analysis.

Meas is the set of measurement schemes that take reaction sequences to a normalised measurement (between 0 and 1) of how 'good' that sequence is.

$$Meas \triangleq React^* \rightarrow [0,1] \tag{8.3}$$

A distribution is a function from a normalised measurement to probability. *Distr* is the set of all such distributions.

$$Distr \triangleq [0,1] \rightarrow [0,1] \tag{8.4}$$

A usage distribution only makes sense if it explicitly includes a measurement scheme. Hence a usage distribution is a pair of measurement scheme and distribution.

$$UD \triangleq Meas \times Distr \tag{8.5}$$

Now we can show how usage distributions relate to the model of use we have proposed. A usage distribution is an approximation of use — a shorthand as it were. Hence a single distribution can describe several uses. This approximation is based on the measurement scheme, the measurement scheme defines equivalences sets of reaction sequences (where an equivalence set is a set of all reaction sequences that have the same measurement), and the distribution defines how likely it is for all those reaction sequences to occur.

The function *equivSet* takes a measurement scheme and a normalised measure between 0 and 1 and returns the set of all reaction sequences that have that measure according to the measurement scheme.

$$\begin{aligned} &equivSet : Meas \times [0,1] \rightarrow \mathcal{P}(React^*) \\ &equivSet(m, val) \mathrel{\hat{=}} \{rs : React^* \mid m(rs) = val\} \end{aligned} \tag{8.6}$$

In words; 'the set of reaction sequences rs is returned such that rs measures val according to the measurement scheme m.'

The predicate *useUD* takes a usage distribution and a use and holds true if the usage distribution correctly describes the use.

$$\begin{aligned} &useUD : UD \times Use \rightarrow \mathbb{B} \\ &useUD((m, d), u) \mathrel{\hat{=}} \\ &\quad \forall val : [0,1] \bullet \exists rss : \mathcal{P}(React^*) \bullet rss = equivSet(m, val) \wedge \\ &\qquad \sum_{rs \in rss} u(rs) = d(val) \wedge \\ &\qquad \exists p : [0,1] \bullet \forall rs : React^* \bullet rs \in rss \Rightarrow u(rs) \approx p \end{aligned} \tag{8.7}$$

In words; 'for every measurement val there is an equivalence set rss of reaction sequences that share that measurement. The distribution defines a probability for all those reaction sequences as being $d(val)$. The total of all the probabilities of the reaction sequences in rss defined by the use should equal the probability defined in the distribution. Furthermore all the reaction sequences in rss should have approximately the same probability.'

This caveat ensuring that all reaction sequences in an equivalence set have roughly the same probability ensures that we cannot propose a use in which all the reaction sequences in an equivalence set have the probability 0 except one which has the probability $d(val)$, which is not

what we are really after.

8.3 ISSL specifications

8.3.1 An abstract syntax for ISSL specifications

An ISSL specification is similar to a reaction style RSSL specification (see section 6.9) except that each reaction is associated with a probability function which gives the probability of that reaction happening next. Pf is the set of all such functions.

$$Pf \mathrel{\hat{=}} React^* \to [0,1] \tag{8.8}$$

$ISSL$ is the set of all ISSL specifications.

$$ISSL \mathrel{\hat{=}} P \times (React \xrightarrow{P} Pf) \tag{8.9}$$

...where if $(init, reacts) : ISSL$ then...

- $init$ is the initial predicate which describes states at time zero,
- $reacts$ is the set of reactions mapped to a probability function describing how likely they are to occur, and
- at any time the probabilities generated by the probability functions must total 1.

$$\forall rs : React^* \bullet \sum_{r \in \mathbf{dom}\ reacts} reacts(r)(rs) = 1$$

Each probability function in an ISSL specification is expressed as a pair of probability functions for the user and interface effect. A 'raw' ISSL specification is one where each reaction is associated with a pair of probability functions.

$$ISSLraw \mathrel{\hat{=}} P \times (React \to (Pf \times Pf)) \tag{8.10}$$

We shall give semantics for ISSL specifications, so we need to show how to convert a raw ISSL specification to an ISSL specification. The function *filter* takes the reactions and probability functions of a raw specification and returns those reactions mapped to single probability functions. Those single functions are the result of 'filtering' the user and interface effects in the raw specification (as described in section 7.4.1).

$$\begin{aligned}
&\mathit{filter} : (\mathit{React} \xrightarrow{p} (\mathit{Pf} \times \mathit{Pf})) \to (\mathit{React} \xrightarrow{p} \mathit{Pf}) \\
&\mathit{filter}(\{r_1 \mapsto (u_1, i_1), \ldots, r_n \mapsto (u_n, i_n)\}) \triangleq \\
&\qquad \{r_j \mapsto \{rs \mapsto (u_j(rs) \times i_j(rs) \times 1/T) \mid \\
&\qquad\qquad T = \sum_{1 \le k \le n} (u_k(rs) \times i_k(rs))\} \mid 1 \le j \le n\}
\end{aligned} \tag{8.11}$$

In words; 'the function *filter* takes n reactions associated with pairs of probability functions. For each of these a reaction associated with a single probability function is returned such that the probability function maps reaction sequences rs to a probability calculated by multiplying the interface effect for rs to the user effect for rs to a normalising value $1/T$ where T is the total of all the multiplied together user and interface effects for rs.'

Converting a raw ISSL specification is simply a case of applying *filter* to its reactions and probability functions. The function *unraw* does this.

$$\begin{aligned}
&\mathit{unraw} : \mathit{ISSLraw} \to \mathit{ISSL} \\
&\mathit{unraw}((\mathit{init}, \mathit{rawReacts})) \triangleq (\mathit{init}, \mathit{filter}(\mathit{rawReacts}))
\end{aligned} \tag{8.12}$$

8.3.2 Semantics for ISSL specifications

Our aim is to define a function that takes an ISSL specification to a use. First we define the function *prob* that calculates the probability of a sequence of reactions occurring.

$$\begin{aligned}
&\mathit{prob} : \mathit{ISSL} \times \mathit{React}^* \to [0, 1] \\
&\mathit{prob}((\mathit{init}, \mathit{reacts}), \langle r_1, \ldots, r_n \rangle) \triangleq \\
&\qquad 0 \text{ if } \langle r_1, \ldots, r_n \rangle \notin \mathit{reactBeh}((\mathit{init}, \mathbf{dom}\ \mathit{reacts})) \\
&\qquad \prod_{1 \le i \le n} \mathit{reacts}(r_i)(\langle r_1, \ldots, r_{i-1} \rangle) \text{ otherwise}
\end{aligned} \tag{8.13}$$

In words; 'if the reaction sequence $\langle r_1, \ldots, r_n \rangle$ is illegal then it has a probability of 0. Otherwise the product of all the individual probabilities is returned. An individual probability (say for reaction r_i) is calculated by extracting the appropriate probability function from the specification $reacts(r_i)$ and then applying the sequence of reactions so far $\langle r_1, \ldots, r_{i-1} \rangle$ to that function.'

Calculating the use described by a ISSL specification is simply a case of mapping all reaction sequences to the probability defined for them by *prob*. The function *useISSL* does this.

$$\begin{array}{l} useISSL : ISSL \rightarrow Use \\ useISSL(spec) \mathrel{\hat{=}} \{rs \mapsto p \mid p = prob(spec, rs)\} \end{array} \tag{8.14}$$

It is interesting to note that the semantics for an ISSL specification define a single use for a specification, whereas the semantics for usage distributions define a collection of uses that are described by a single distribution. If we consider usage distributions to be requirements for use and that ISSL is a specification for use then we have sensible relation between the two according to our semantics. A usage distribution is an approximation of use and therefore defines a space of possible uses whereas an ISSL specification defines precisely one use. The predicate *consistent* holds true if the use defined by an ISSL specification is one that is valid for a usage distribution.

$$\begin{array}{l} consistent : UD \times ISSL \rightarrow \mathbb{B} \\ consistent(ud, spec) \mathrel{\hat{=}} useUD(ud, useISSL(spec)) \end{array} \tag{8.15}$$

We can achieve this neat relationship between usage distributions and ISSL specifications because of the simple nature of the underlying model. If we were using the more involved model that maps activities to probabilities then an ISSL specification would describe a set of possible uses. This is because of the argument that one activity can be caused by several reaction sequences, and hence it would not be possible to calculate one definitive probability for an activity.

Theoretically we would define consistency between distributions and specifications as holding if the uses described by the specification are a sub-set of those described by the distribution. However, although mathematically neat this may be impractical. By their nature all the models and concepts we use in considering the use of a system are approximate, especially things like

measurement schemes, and hence the comparison of sets of uses should be equally approximate. Showing this sort of sub-set consistency would only be sensible if the usage distributions and the user and interface effects we use in the analysis are precise. In the absence of such precision we can only expect approximate comparisons — consistency would be more likely to be based on there being a significant intersection between the uses described by a distribution and a ISSL specification. What is 'significant' is open to debate.

By employing the simpler model of use we do not need to worry so much about these matters.

8.4 A regular grammar-like notation

In the previous chapter we used a regular grammar notation to describe the optimal behaviour of a system and we also used it in defining probabilistic functions for the user and interface effects.

This notation describes orderings of reactions and is defined inductively. A null sequence (denoted Λ) or a single reaction is the base case and if n is a natural number, *tvf* is a truth valued function and R, R_1 and R_2 are valid regular sentences then so are...

$$R_1 R_2 \qquad R_1 + R_2 \qquad R^n \qquad R^* \qquad R^x \bullet tvf$$

(The last sentence describes a sentence repeated x times where x is a natural number which satisfies the truth-valued function. *e.g.* $R^x \bullet 1 \leq x \leq 10$ is the repetition of R between 1 and 10 times.)

Formally...

$$RG ::= \Lambda \mid React \mid RG\ RG \mid RG + RG \mid RG^n \mid RG^* \mid RG^x \bullet TVF \tag{8.16}$$

The predicate *rg* takes a sentence from RG and a sequence of reactions and holds true if that sequence is correctly described by the sentence.

$$
\begin{aligned}
&rg : RG \to React^* \to \mathbb{B} \\
&\quad rg[\![\Lambda]\!]rs \triangleq rs = \langle\rangle \\
&\quad rg[\![r]\!]rs \triangleq rs = \langle r \rangle \\
&\quad rg[\![R_1 R_2]\!]rs \triangleq \exists rs_1, rs_2 : React^* \bullet rg[\![R_1]\!]rs_1 \wedge rg[\![R_2]\!]rs_2 \wedge \\
&\qquad\qquad rs = rs_1 \frown rs_2 \\
&\quad rg[\![R_1 + R_2]\!]rs \triangleq rg[\![R_1]\!]rs \vee rg[\![R_2]\!]rs \\
&\quad rg[\![R^0]\!]rs \triangleq rs = \langle\rangle \\
&\quad rg[\![R^{n+1}]\!]rs \triangleq \exists rs_1, rs_2 : React^* \bullet rg[\![R]\!]rs_1 \wedge rg[\![R^n]\!]rs_2 \wedge \\
&\qquad\qquad rs = rs_1 \frown rs_2 \\
&\quad rg[\![R^*]\!]rs \triangleq \exists n : \mathbb{N} \bullet rg[\![R^n]\!]rs \\
&rg[\![R^x \bullet tvf]\!]rs \triangleq \exists n : \mathbb{N} \bullet rg[\![R^n]\!]rs \wedge tv[\![tvf, \langle n \rangle]\!]
\end{aligned}
\tag{8.17}
$$

In words;

- 'the null case — rs is the null sequence,
- the base case — a reaction describes a singleton list containing that reaction,
- sequence — rs can be split into two sub-sequences the first of which is described by R_1 and the second sub-sequence is described by R_2,
- disjunction — rs is described by either of the sentences R_1 or R_2,
- repetition 0 times — rs is the null sequence,
- repetition n+1 times — rs can be split into two sub-sequences the first of which is described by the sentence R and the second is described by R repeated n times,
- repetition arbitrarily — R repeated an arbitrary (but finite) number of times describes rs, and
- repetition a defined number of times — R is repeated n times where n satisfies the truth-valued function tvf.'

These regular grammar sentences are used to describe a constraining of the set of reactions sequences that are legal for a specification. For example the sentence *WPoptimal* (formula 7.16) defined in the previous chapter defines a sub-set of the legal reaction sequences of the word processor specification *WP* (figure 7-1).

The function *rgBeh* takes a reaction style specification and a regular grammar sentence and returns the set of reaction sequences that are legal for the specification and described by the sentence.

$$\begin{array}{l} rgBeh : RG \times RSSLreact \rightarrow \mathcal{P}(React^*) \\ rgBeh(R, spec) \triangleq \{rs : React^* \mid rs \in reactBeh(spec) \wedge rg[\![R]\!]rs\} \end{array} \tag{8.18}$$

In words; 'the set of reaction sequences *rs* is returned that are in the set of legal reaction sequences for *spec* and are correctly described by *R*.'

8.5 Probability functions

Recall that a probability function is a function from reaction sequences to probability.

$$Pf : React^* \xrightarrow{p} [0, 1]$$

We describe probability functions as a collection of regular grammar sentences and associated probabilities as demonstrated in figure 7-8. All the reaction sequences described by a sentence will be mapped to the probability associated with it. The abstract syntax for the probability function definitions is a set of paired regular grammar sentences and probabilities.

$$Pfdef \triangleq \mathcal{P}(RG \times [0, 1]) \tag{8.19}$$

If $pfd : Pfdef$ then all the sentences in *pfd* should describe distinct sets of reaction sequences. The following description of a probability function is not acceptable...

$$\begin{array}{l} (A + B) \Rightarrow 0.5 \\ (B + C) \Rightarrow 0.25 \end{array} \tag{8.20}$$

...as it defines two values for the sequence $\langle B \rangle$. Formally, the intersection of all the sets of reaction sequences described by all the sentences in a probability function definition is the empty set.

$$\forall pfd : Pfdef \bullet (\bigcap_{(R,p)\in pfd} \{rs : React^* \mid rg[\![R]\!]rs\}) = \emptyset \tag{8.21}$$

In words; 'the intersection of all set of reaction sequences described by the sentences R in a probability function definition is null.'

These descriptions are converted to a probability function by evaluating the regular grammar sentences using the function rg defined in the previous section and mapping them to the given probabilities. The function pf performs this conversion.

$$\begin{array}{l} pf : Pfdef \rightarrow Pf \\ pf(pfd) \mathrel{\hat{=}} \{rs \mapsto 0 \mid rs : React^*\} \\ \qquad\qquad \oplus \\ \qquad \{rs \mapsto p \mid \exists R : RG \bullet (R, p) \in pfd \wedge rg[\![R]\!]rs\} \end{array} \tag{8.22}$$

In words; 'the probability function which maps all reaction sequences to 0 is overwritten with a partial function from reaction sequences described by sentences R in pdf to the probability R is paired with. (Note that probability functions are total, hence the need to overwrite the 'null' probability function to ensure totality.)'

8.6 Discussion

This chapter has formalised the ideas introduced in chapter 7. We have used a very simple model of use, namely a distribution of probability over sequences of reactions. In doing so we have captured fairly elegantly the distinction between usage distributions and ISSL specifications; namely that usage distributions describe a collection of possible uses whereas ISSL specifications describe precisely one use.

We have alluded to the possibility of using a more complicated model of use which is a distribution of probability over real-time activities. We can now look in a little more detail at this model, how we might go about formalising it and why this formalisation is considerably more complicated than the one introduced in this chapter.

8.6.1 An uncountably large space of activities

Because an activity is indexed by real time, a behaviour will typically contain uncountably many activities. Asking the probability of a single event in an uncountable space has little meaning. Consider the probability of obtaining a single point from a continuous space of real numbers. The probability of obtaining 7.5 is zero (or at most infinitesimally small) and so it only makes sense to consider the probability of obtaining a value in a range of real numbers, say 7.4 – 7.6.

Now consider the following specification.

$$increm \mathrel{\hat{=}} x = 0 \wedge \Box[justAddOne] \tag{8.23}$$

$$\begin{aligned} justAddOne \mathrel{\hat{=}} & \textbf{input} : x \\ & \textbf{output} : x \\ & \textbf{enabled\_by} : x = 0 \\ & \textbf{outcome} : x' = x + 1 \wedge \mathrm{t}' \leq \mathrm{t} + 2 \end{aligned} \tag{8.24}$$

Assuming that time is measured in one second units this specifies a system that adds one to x in the first two seconds and then stops. Even this simple specification describes an uncountably large set of activities. The change from $x = 0$ to $x = 1$ can take place at (say) time 0.5, or time 1, or any time in between. As time is real then there are an uncountable number of time values in between 0.5 and 1 that the change can take place in, hence the behaviour space is uncountably large.

If even a very simple specification such as *increm* has an uncountably large behaviour space then it is not unreasonable to assume that the majority of specifications will have too. Indeed unless a specification is exact about the time for every state change then a specification must have an uncountably large behaviour space.

To overcome this problem we would perhaps approximate behaviours by only considering behaviours where state discontinuities occur (if they occur at all) at regular intervals.

8.6.2 Infinitely long activities

An ISSL specification applies probabilities to reactions. An activity that represents an infinite number of reactions has a probability of zero. Therefore we need to add a further level of approximation to activities by considering only those that represent a finite number of reactions.

Typically a specification in the reaction style will describe plenty of such finite legal activities. If we were considering specifications with perpetually enabled obligatory computations then there would be no legal finite activities. As we are only considering reaction style specifications then the obligatory computations are only enabled by environment computations, hence there will be legal finite activities.

The problems with probabilistic non-determinism in infinite activities are addressed in [68], but the application of that work to this is beyond our scope.

8.6.3 The relationship between reaction sequences and activities

The relationship between reaction sequences and activities is not a function. The probability of an activity occurring is calculated from the reaction sequence, so a single activity will have several probabilities associated with it. This means that the function *useISSL* that takes ISSL specifications to single uses cannot be defined — we need to define a function that takes ISSL specifications to sets of possible uses. This adds a level of complexity to the comparison between the uses described by usage distributions and ISSL specifications and it is debatable how we should deal with that comparison. Because of the approximate nature of many of the entities we have used (notably measurement schemes, and user and interface effects) then we must be aware that we will be putting approximate information into our models and getting approximate results out. In this light it is not too sensible to be extremely rigorous in comparing the evaluations of different models.

8.6.4 Conclusion

This chapter has described formally ways of expressing the requirements and specifications of system use. We do not claim these to be only ways of doing so, but we have linked them to the RSSL specification language we presented in earlier chapters. Indeed a different way of specifying use is presented in [24].

We propose that there are two sets *REQ* and *SPEC* which are the sets of all mathematical entities that may be used to describe use requirements and specifications respectively. *UD* is a sub-set of *REQ* and *ISSL* is a sub-set of *SPEC*. We can also propose two semantic functions for evaluating entities in these sets.

$$\mathcal{R}{:}REQ \rightarrow \mathcal{P}(Use) \tag{8.25}$$

$$\mathcal{S}{:}SPEC \rightarrow \mathcal{P}(Use) \tag{8.26}$$

Although *useUD* and *useISSL* do not strictly fit into these template signatures, it would not be difficult to redefine them so they did so. When choosing entities from *REQ* and *SPEC* as tools for describing use, it is best to choose entities that allow fairly approximate descriptions for requirements and less approximate descriptions from *SPEC*. This is trivially the case for *UD* and *ISSL* because *useUD* defines which uses are correct for a given usage distribution and *useISSL* defines which single use is correct for a given specification.

Chapter 9

Some examples of our technique in use

In this chapter we apply our approach to some examples and see if we can expose some strengths and weaknesses. Firstly we attempt to express some PIE-like properties (section 9.1), then we refine part of the specification of the word processor presented in chapter 7 (section 9.2). Then we take a well-known psychological effect and show how we can use it to guide the design of an interactive system (section 9.3). We conclude with a discussion of the examples (section 9.4).

9.1 Expressing PIE-like properties

In this section we show that we can express PIE-like usability properties [41] within our approach. We do this to show that we can link in the usability properties approach to our work and that we can describe systems (or system requirements) in a way as abstract as the PIE model.

We shall express the properties in the simple temporal logic we use for expressing system requirements. We contend the usability properties proposed by Dix and others are about requirements for a system rather being a model of the system itself, we therefore express them in our 'requirements language'. We also note from [1] the elegant way that properties can be dealt with by TLA, *i.e.* if P is a property and *sys* is a model of a system then showing that the system fulfills that property is simply a case of showing that...

$$sys \Rightarrow P$$

We contend that as our system modelling approach is based on TLA we would be able to use this method too.

9.1.1 Observability

The observability properties are straight forward to formulate within our approach and in fact are close to being a word-for-word translation from [38].

The state space for the required system consists of the internal effect e, the display d and the result r.

$$e{:}E \tag{9.1}$$
$$d{:}D \tag{9.2}$$
$$r{:}R \tag{9.3}$$

The result and display are a function of the effect; there are two functions that take effect to display and result.

$$\mathit{display}{:}E \to D \tag{9.4}$$
$$\mathit{result}{:}E \to R \tag{9.5}$$

There is a safety property that ensures that the display and result are always[1] a correct representation of the effect.

$$\Box(d = \mathit{display}(e) \wedge r = \mathit{display}(e)) \tag{9.6}$$

An observable system is one in which it is possible to determine what the result would be from the display, *i.e.* there is a function *obs* from displays to results which is related to the two functions *display* and *result*. Strict observability says that the result and display are always linked by this function.

[1] Actually this is too strong — we need an infinitely fast machine to implement this safety property. See [23] for a way of getting round this problem.

$$strictObserve \triangleq \exists obs : D \rightarrow R \bullet \forall e' : E \bullet obs(display(e')) = result(e') \wedge \Box(obs(d) = r) \quad (9.7)$$

All the caveats that apply to the PIE formalisation of this property also apply here. *obs* should be a sensible function — we can propose a function *obs* that takes word processed documents in English on the screen to printed documents in Greek. In this case the function *obs* exists, is valid and holds between d and r but we would be hard pressed to argue that it captures a WYSIWYG system. Furthermore this property is too strong for any result that is too large to fit on one screen. Dix gets round this by proposing that the display is in fact a sub-set of the observable area, that the observable area is related to the result by *obs* and the user can get to any part of the observable area by a 'passive' collection of commands.

Note in our formulation saying 'there exists a function that always holds between d and r' is *not* the same as saying that 'there always exists a function that holds between d and r'. In this latter case *obs* could change during run-time, so one time we may print a document and get it in English and another time get it in Greek.

We can loosen the observability property temporally perhaps by saying that *obs* always eventually holds between d and r, so the system can always get into a state where the result is observable from the screen. Again this is too strong — we only want the system to get in such a state if requested by the user. Assume there is some predicate *obsReq* that holds when the user has requested the device to show the entire result. (This equates to selecting 'print preview' in a drawing package.)

$$\begin{aligned} requestObserve \triangleq \exists obs : D \rightarrow R \bullet \forall e' : E \bullet obs(display(e')) = result(e') \wedge \\ \Box(obsReq \Rightarrow \Diamond(obs(d) = r)) \end{aligned} \quad (9.8)$$

9.1.2 Reachability

Reachability captures that it is never possible to get the system into a dead-end, *i.e.* every state can be eventually got to from any other state. If we assume that e is the underlying state then

we might express reachability as follows...

$$strictReach \triangleq \exists x \bullet \Diamond(e = x) \Rightarrow \Box \Diamond(e = x) \tag{9.9}$$

In words; 'if the system gets into a state where $e = x$ at all then it must always reaches that state in the future.'

This is not what we want though. What we should be saying is that if the system *can* get to a state where $e = x$ then it *can* always get into that state. Hence we can argue that temporal logic is not quite the language for expressing reachability as it does not clearly distinguish between 'can' and 'must'.

However if we assume that a liveness property is given which captures the fact that the environment makes requests that the kernel must eventually respond to.

$$live \triangleq \Box(req \Rightarrow \Diamond resp) \tag{9.10}$$

Now we could define reachability as follows...

$$reach \triangleq \exists x \bullet \Diamond(e = x) \Rightarrow \Box(req \Rightarrow \Diamond(e = x)) \tag{9.11}$$

In words; 'if the system gets to a state where $e = x$ then it is always the case that when the environment makes an appropriate request then the system will get the system state to where $e = x$.'

Note that *req* need not correspond in a one to one way to invocations in the system specification, so *req* may be specified as a collection of invocations.

9.1.3 Conclusions about the PIE properties

We have expressed observability quite elegantly in our framework and see no reason why we cannot capture the caveats to these properties that Dix formalises. Capturing reachability was a bit more tricky and it is debatable that our definition of reachability is strictly equivalent to the PIE version — it depends on how we define the liveness property. It would probably be more sensible to define reachability on a system specification where the reactions are defined explicitly. Defining reachability would then be a case of expressing whether there is at least one

sequence of reactions that can get the system from one state to any other.

9.2 Refining the word processing example

Recall the word-processor example from chapter 7 — the model defined three reactions for editing the text, formatting the text and performing disk operations. In this section we refine the *edit* reaction and the state model.

The first matter to consider is the possible concurrency in the system. It is important to control the concurrency, because we want characters to be inserted into the text in the same order as the commands to insert them were issued. We do not want a system that inserts all the right letters but not necessarily in the right order. To achieve this we constrain the concurrency in the system so that only one reaction can occur at a time. There is a boolean flag *free* which is set false once any invocation occurs and is not set true until its response has completed. Invocations are not enabled while *free* is false.

$$free : \mathbb{B} \tag{9.12}$$

This approach may be considered overly restrictive. In section 9.2.4 we sketch out a specification for a system which allows for buffered commands offered by the environment and a kernel that processes the buffer in a batch-like way as is done in distributed systems.

Recall in chapter 7 we left the state space to be fairly undefined...

$$\begin{aligned} inDev &: \ldots \\ text &: \ldots \\ formatting &: \ldots \\ disk &: \ldots \end{aligned}$$

The reaction *edit* consists of an invocation that does something (that is undefined) to *inDev* and a response that changes *text* and possibly changes *formatting*.

9.2.1 Refining the state space

In this refinement we do not refine the model of *inDev*, nor do we worry about *disk*. To refine *text* and *formatting* we define a set of characters and a set of formats as follows...

$$Form \triangleq \{\mathsf{normal}, \mathsf{bold}, \mathsf{italic}, \ldots\} \tag{9.13}$$

$$Char \triangleq \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \ldots, \mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots, 0, 1, 2, \ldots\} \tag{9.14}$$

We define *text* to be four sequences of $\mathcal{C}har$. The first sequence is all the text before the cursor, the second is all the text that is selected, the third is all the text after the cursor and the fourth is all the text on the paste board. If there is no selected text then the second sequence is null.

$$text{:}(Char^* \times Char^* \times Char^* \times Char^*) \cup \{\mathsf{Null}\} \tag{9.15}$$

The token value Null is included to denote the value of text before anything has been loaded from the disk.

The formatting is similar — each character in *text* has a set of formats associated with it. So *formatting* is four sequences of sets of formats, each sequence corresponding to the sequence in *text* — the first sequence is all the formatting before the cursor, the second all the formatting of the selected text, the third all the formatting after the cursor and the fourth all the formatting on the paste board.

$$formatting{:}(\mathcal{P}(form))^* \times (\mathcal{P}(form))^* \times (\mathcal{P}(form))^* \times (\mathcal{P}(form))^* \cup \{\mathsf{Null}\} \tag{9.16}$$

We can furthermore propose a safety property that expresses that the corresponding sequences in *text* and *formatting* are the same size...

$$\Box(\exists st, cur, end, pb, stf, curf, endf, pbf \bullet \\ text = (st, cur, end, pb) \wedge \\ formatting = (stf, curf, endf, pbf) \wedge \\ |st| = |stf| \wedge |cur| = |curf| \wedge \\ |end| = |endf| \wedge |pb| = |pbf|) \tag{9.17}$$

...so that there is the same amount of formatting before the cursor as there is text and so on.

9.2.2 Refining the reaction *edit*

Recall from section 7.4.1 we suggested that we refine the reaction *edit* into five decomposed reactions *insert*, *delete*, *moveCursor*, *copy* and *paste*. Each of these reactions consists of invocations and responses as follows...

$$insert \mathrel{\hat{=}} insertI \fatsemi insertR \tag{9.18}$$
$$delete \mathrel{\hat{=}} deleteI \fatsemi deleteR \tag{9.19}$$
$$moveCursor \mathrel{\hat{=}} moveCursorI \fatsemi moveCursorR \tag{9.20}$$
$$copy \mathrel{\hat{=}} copyI \fatsemi copyR \tag{9.21}$$
$$paste \mathrel{\hat{=}} pasteI \fatsemi pasteR \tag{9.22}$$

The invocations

The invocations still describe the alteration of the input device in some way, but as yet we are not at a low enough level of abstraction to define the actual alterations to the input device. The invocations are not enabled until *free* is true and *copyI* is not enabled unless there is some selected text, *deleteI* is not enabled if there is nothing before the cursor and *pasteI* is not enabled if there is nothing on the paste board. All this is shown in figure 9-1.

The response *insertR*

All the responses have similar forms. The read clause extracts the parts of the state that we wish to manipulate and passes them to the private space. The process clause manipulates the

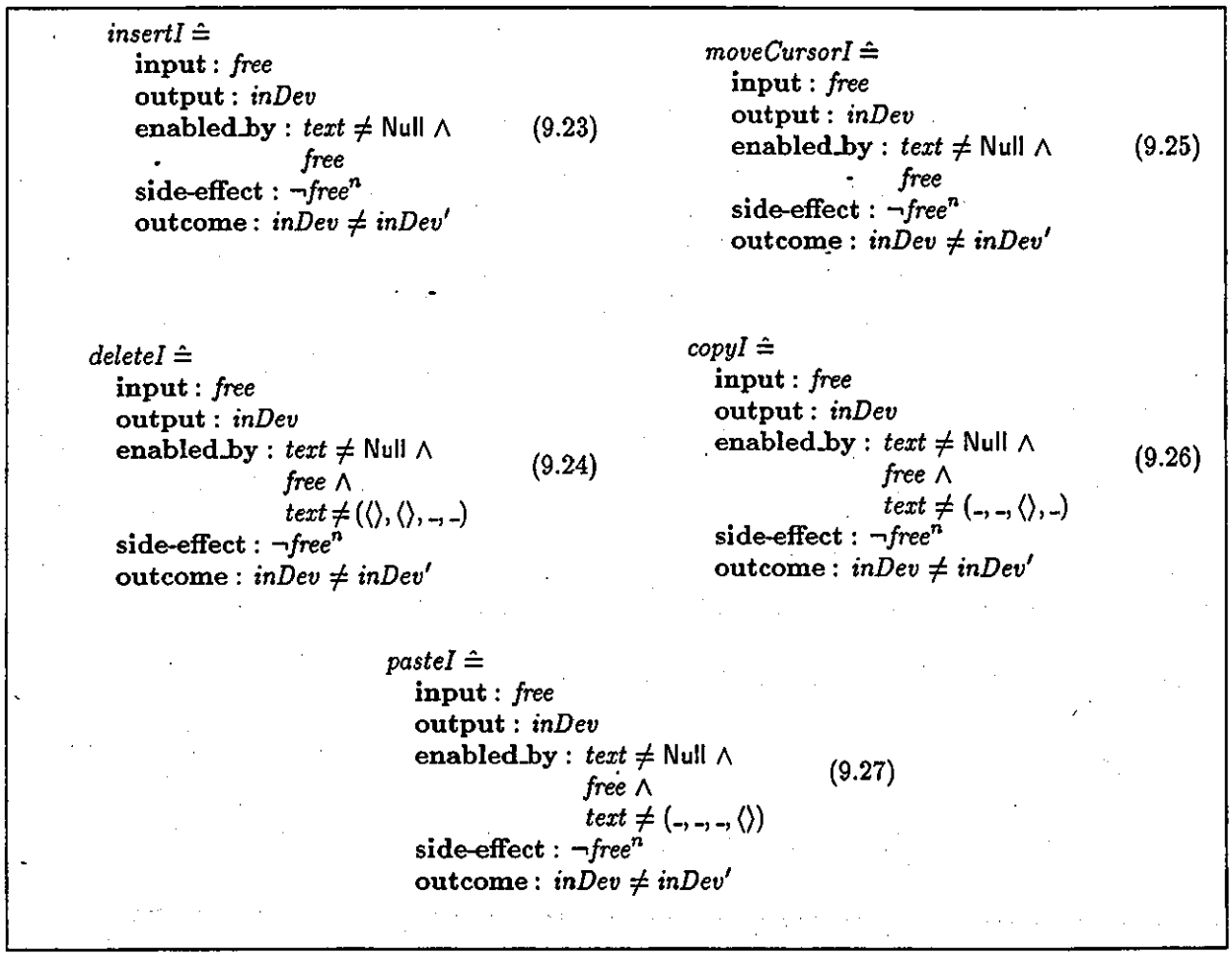


Figure 9-1: Refined invocations

private space and then the write clause writes the manipulated parts of the state back to the public space.

In the following formalisations we use the following denotations to split up *text* and *formatting*...

$$text = (st, cur, end, pb)$$

$$formatting = (stf, curf, endf, pbf)$$

insertR adds a character to the end of the first sequence and deletes any text that is selected. This new character either assumes the formatting of the last character in the first sequence if there is no text selected or it assumes the formatting of the first character of the selected text.

$$
\begin{aligned}
insertR \mathrel{\hat{=}} & \textbf{input} : text, formatting \\
& \textbf{output} : free, text, formatting \\
& \textbf{private} : st, cur, stf, curf \\
& \textbf{invoked\_by} : insertI \\
& \\
& \textbf{read} : \exists end, pb, endf, pbf \bullet \\
& \qquad text = (st^n, cur^n, end, pb) \wedge \\
& \qquad formatting = (stf^n, curf^n, endf, pbf) \\
& \\
& \textbf{process} : \exists c \bullet st^p = st^n \frown \langle c \rangle \wedge cur^p = curf^p = \langle\rangle \wedge \\
& \qquad curf^n = \langle\rangle \Rightarrow stf^p = stf^n \frown \langle lt(stf^n) \rangle \wedge \\
& \qquad curf^n \neq \langle\rangle \Rightarrow stf^p = stf^n \frown \langle hd(curf^n) \rangle \\
& \\
& \textbf{write} : text' = (st^p, cur^p, end, pb) \wedge \\
& \qquad formatting' = (stf^p, curf^p, endf, pbf) \wedge \\
& \qquad free'
\end{aligned}
\tag{9.28}
$$

In words; 'the computation *insertR* takes input from *text* and *formatting* and outputs to *text*, *formatting* and *free*. It has internal space consisting of *st*, *cur*, *stf* and *curf* to which the values of the text before the cursor, the selected text, the formatting before the cursor and the formatting of the selected text are copied respectively.

The read phase copies values from the public space to the internal space and the write phase copies the processed versions back to the external space. The write phase also sets *free* to true. Note that the text and formatting after the cursor and the paste board are unaffected.

The process phase adds some character *c* to the text before the cursor and deletes any text that is selected. The new character assumes the formatting of the character that was previously immediately before the cursor, or the formatting of the first character of the selected text.'

The response *deleteR*

The response *deleteR* is similar to *insertR*. It does not effect the text after the cursor or the paste board. If there is selected text then it is deleted. If there is no selected text then the character immediately before the cursor is deleted. (Recall that the invocation *deleteI* is not enabled if there is nothing to delete.)

$$
\begin{aligned}
deleteR \hat{=}\ & \textbf{input} : text, formatting \\
& \textbf{output} : free, text, formatting \\
& \textbf{private} : st, cur, stf \\
& \textbf{invoked\_by} : deleteI \\
& \\
& \textbf{read} : \exists curf, end, pb, endf, pbf \bullet \\
& \qquad text = (st^n, cur^n, end, pb) \wedge \\
& \qquad formatting = (stf^n, curf, endf, pbf) \qquad (9.29) \\
& \\
& \textbf{process} : cur^n = \langle\rangle \Rightarrow st^p = ft(st^n) \wedge stf^p = ft(stf^n) \wedge \\
& \qquad cur^n \neq \langle\rangle \Rightarrow st^n = st^p \wedge stf^n = stf^p \\
& \\
& \textbf{write} : text' = (st^p, \langle\rangle, end, pb) \wedge \\
& \qquad formatting' = (stf^p, \langle\rangle, endf, pbf) \wedge \\
& \qquad free'
\end{aligned}
$$

In words; 'the appropriate parts of the public space are copied to *st*, *cur* and *stf*. If *cur* is empty then the character (and its formatting) before the cursor is removed, whereas if *cur* is not empty then *st* and *stf* are unchanged. The private space is then copied back to the public space and the selected text is set to being empty.'

The response *moveCursorR*

moveCursorR simply reorganises (in some way) the first three sequences in *text* and *formatting* without losing any of their information or changing their order.

$$
\begin{aligned}
moveCursorR \hat{=}\ & \\
& \textbf{input} : text, formatting \\
& \textbf{output} : free, text, formatting \\
& \textbf{private} : st, cur, end, stf, curf, endf \\
& \textbf{invoked\_by} : moveCursorI \\
& \\
& \textbf{read} : \exists pb, pbf \bullet \\
& \qquad text = (st^n, cur^n, end^n, pb) \wedge \\
& \qquad formatting = (stf^n, curf^n, endf^n, pbf) \qquad (9.30) \\
& \\
& \textbf{process} : st^n \frown cur^n \frown end^n = st^p \frown cur^p \frown end^p \wedge \\
& \qquad stf^n \frown curf^n \frown endf^n = stf^p \frown curf^p \frown endf^p \wedge \\
& \qquad |st^p| = |stf^p| \wedge |cur^p| = |curf^p| \wedge |end^p| = |endf^p| \\
& \\
& \textbf{write} : text' = (st^p, cur^p, end^p, pb) \wedge \\
& \qquad formatting' = (stf^p, curf^p, endf^p, pbf) \wedge \\
& \qquad free'
\end{aligned}
$$

In words; 'the process phase reorganises the text before the cursor, the selected text and the text after the cursor in some way. No text or formatting is lost and the formatting is reorganised in the same way that the text is, so there is as much formatting before the cursor as there is text *etc.*'

The response *copyR*

The *copyR* response copies the selected text and formatting to the paste board.

$$
\begin{aligned}
copyR \mathrel{\hat{=}} & \textbf{input} : text, formatting \\
& \textbf{output} : free, text, formatting \\
& \textbf{private} : cur, pb, curf, pbf \\
& \textbf{invoked\_by} : copyI \\
& \\
& \textbf{read} : \exists st, end, stf, endf \bullet \\
& \qquad text = (st, cur^n, end, pb^n) \wedge \\
& \qquad formatting = (stf, curf^n, endf, pbf^n) \\
& \\
& \textbf{process} : pb^p = cur^n \wedge pbf^p = curf^n \wedge \\
& \qquad cur^n = cur^p \wedge curf^n = curf^p \\
& \textbf{write} : text' = (st, cur^p, end, pb^p) \wedge \\
& \qquad formatting' = (stf, curf^p, endf, pbf^p) \wedge \\
& \qquad free'
\end{aligned}
\tag{9.31}
$$

The response *pasteR*

pasteR copies all the text and formatting from the paste board to before the cursor position and removes all the selected text.

$$
\begin{aligned}
pasteR \hat{=}\ & \textbf{input} : text, formatting \\
& \textbf{output} : free, text, formatting \\
& \textbf{private} : st, pb, stf, pbf \\
& \textbf{invoked\_by} : pasteI \\
& \\
& \textbf{read} : \exists cur, end, curf, endf \bullet \\
& \qquad text = (st^n, cur, end, pb^n) \wedge \\
& \qquad formatting = (stf^n, curf, endf, pbf^n) \qquad (9.32) \\
& \\
& \textbf{process} : st^p = st^n \frown pb^n \wedge stf^p = stf^n \frown pbf^n \wedge \\
& \qquad pb^n = pb^p \wedge pbf^n = pbf^p \\
& \\
& \textbf{write} : text' = (st^p, \langle\rangle, end, pb^p) \wedge \\
& \qquad formatting' = (stf^p, \langle\rangle, endf, pbf^p) \wedge \\
& \qquad free'
\end{aligned}
$$

9.2.3 Summary

We have shown how we can refine a specification by refining the state space and decomposing computations. Furthermore we have demonstrated the long-hand computation style which makes more detailed computations clearer. In each computation the relevant parts of the public space are copied to the private space, processed and copied back to the public space.

9.2.4 A sketch of a specification of a distributed word processor

The specification we have shown above is locked into sequential behaviour. This is fine for single user systems, but would be less suitable for a distributed editor where several users can concurrently edit a document.

Instead of starting by breaking the reactions into edits, formats and disk operations we would group them all together into one single reaction consisting of an invocation that adds commands to a pool and a response that takes commands from the pool and processes them. The specification is therefore similar to the specification we used to introduce RSSL in chapter 5.

The invocation passes commands into a pool of as yet unprocessed commands along with a time 'stamp' denoting the time at which the command was issued. The response selects outstanding commands from the pool and processes them in the order of their time stamps.

There needs to be some resolution strategy for commands with the same time stamp that make contradictory changes to the state. It is possible to also define such things as user precedence so that commands issued by a user with a high precedence override contradictory commands issued by low precedence users.

There is no necessity for the response to process commands one by one — we can define a more batch-like process where several commands are processed at once and state is updated with the cumulative effect of several commands at once. This can minimise the need for a high band-width of information passed from the kernel to the environment, which can be useful if the system is distributed over a wide geographic area.

9.3 The 'trailing sub-goals' problem

The 'trailing sub-goals' problem is an interesting effect to be found in several interactive systems. A user's task can be characterised by the achievement of a goal. In achieving this goal the user may initiate sub-tasks, but if the user does not complete the all the sub-tasks before completing the main task then these sub-tasks can be easily forgotten and not completed.

Consider an automatic teller machine — the user's main goal in using such a machine is to withdraw cash. In order to do so the user must initiate a sub-goal to insert and remove the bank card. In early implementations of some teller machines the cash was dispensed *before* the card was returned and users had a considerable tendency to take the cash and walk away having forgotten the card. The problem here is that the main goal of obtaining cash is satisfied before the sub-goal of removing the card and hence that sub-goal is easily forgotten.

Here we shall discuss another variant of this problem, 'the unselected widow' problem. This problem was analysed at the research symposium of CHI94 by four research teams to see what light could be shed on the problem by different HCI approaches [117]. In a similar way we shall apply our approach to the problem and see what light we can shed.

9.3.1 A statement of the problem

The problem is described by [117] as follows...

'A user is composing a message in window A of a multi-window interface and needs to consult a timetable in window B. The user selects window B, finds the required information and then continues to type the message, forgetting to reselect window A. In these circumstances the typed input goes into the wrong window, which may or may not have disastrous consequences. The error is frequent and persistent and probably occurs at least as much for 'experts' as for 'novices'. Why does it happen and how can the interface be re-designed so as to minimise its occurrence and/or mitigate its effects?'

Why this happens is not our concern, we view the problem from the standpoint of a designer who knows it happens, maybe even knows how likely it is to happen and wants to design an interface to reduce that likelihood.

9.3.2 A system specification

Our system model is very simple — it abstracts away from the task altogether, simply describing a system where some reaction is invoked to perform some work, some reaction is invoked that completes the task and some reaction which moves the focus from one window to another. These reactions are *work*, *finish* and *move* respectively. Given some sensible initial property the system specification is...

$$uwp \triangleq init \wedge \Box\langle work \vee finish \vee move\rangle \tag{9.33}$$

(*uwp* stands for 'unselected window problem').

We define the optimal behaviour to be...

$$uwpOptimal = ((work^*)finish\ move)^* \tag{9.34}$$

In words; 'an optimal user repeatedly does work, finishes the task and then moves to another window and repeats the process.'

The problem stems from the fact that a user is likely to miss out the *move* reaction. We shall discuss whether it is possible to overcome this problem is a purely functional way by in some way linking *finish* to *move* so that the device can detect a *finish* and do the *move* automatically,

thus negating the problem altogether. In many cases this will not be possible to we look at some interface solutions to the problem, that reduce its occurrence if not wholly negating it.

9.3.3 Looking at the problem functionally

If we can get the device to recognise a *finish* reaction then we can overcome the problem. Indeed this is how the problem was cured for the early teller machines we described earlier — the removal of cash from the machine can be easily recognised as the end of the task and therefore we can design a machine that does not allow the user to complete the task until all the sub-tasks are complete. *i.e.* the cash is dispensed *after* the card.

However in a general purpose interactive machine it is likely that it will be impossible to discriminate *work* from *finish*. There are a few tricks that we can use in order to determine whether the user has completed their task — several interfaces have windows that once opened cannot be deselected and close automatically once a 'Done' button is pressed. In this case the user cannot progress to another task until the 'Done' button is pressed and hence the *finish* reaction can be discriminated and *move* occurs automatically. The problem with this approach is that a degree of user freedom is lost because the user cannot perform sub-tasks in different windows.

When the task that is done in a window is well defined, for example selecting file names, then there is good chance that we can get the device to recognise *finish* and we can therefore specify a 'get-around' for the unselected window problem. Generally these get-arounds can restrict user freedom therefore they should be used with caution.

9.3.4 Looking at the problem from the interface perspective

Let us assume that we are defining a general purpose system and therefore cannot use functional remedies to the problem. We therefore need to define interfaces that are going to bias the behaviour to the optimal.

Use requirements

First of all let us define a very simple measurement scheme m which captures how good a given reaction sequence is. As we are *only* worrying about the unselected window problem then we

only need a very simple scheme that returns 0 if the sequence is optimal or 1 otherwise.

$$\begin{array}{l} m : Meas \\ m(rs) \triangleq 0 \text{ if } rs \in rgBeh[\![uwpOptimal, uwp]\!] \\ \qquad\quad 1 \text{ otherwise} \end{array} \tag{9.35}$$

Then we express the use requirements in terms of a usage distribution shown in figure 9-2. These requirements say that users make the unselected window mistake only 10% of the time. This distribution d is defined below...

$$\begin{array}{l} d : Dist \\ d \triangleq \{0 \mapsto 0.9, 1 \mapsto 0.1\} \end{array} \tag{9.36}$$

...and overall the the usage distribution ud is the pairing of the measurement scheme and distribution.

$$ud = (m, d) \tag{9.37}$$

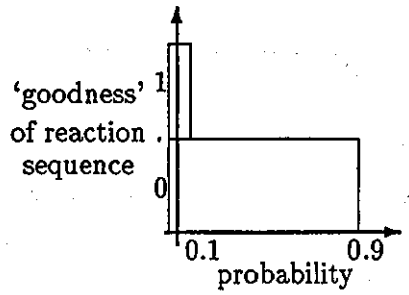


Figure 9-2: A usage distribution for the unselected window problem

The interface specification

We propose a user effect for the system, then we suggest an interface effect so that when we filter the two together we get a system the use of which matches that of the usage distribution ud.

A partial specification of the reactions for the system is given in figure 9-3. Each reaction is split into their constituent invocations and responses and the invocations include pointers to

the probability functions for the user and interface effects.

$$work \mathrel{\hat{=}} workI \fatsemi workR \quad (9.38)$$
$$finish \mathrel{\hat{=}} finishI \fatsemi finishR \quad (9.39)$$
$$move \mathrel{\hat{=}} moveI \fatsemi moveR \quad (9.40)$$

$workI \mathrel{\hat{=}}$
$\vdots$
user : pu_{work}
interface : pi_{work}
$\vdots$ (9.41)

$finishI \mathrel{\hat{=}}$
$\vdots$
user : pu_{finish}
interface : pi_{finish}
$\vdots$ (9.42)

$moveI \mathrel{\hat{=}}$
$\vdots$
user : pu_{move}
interface : pi_{move}
$\vdots$ (9.43)

Figure 9-3: A partial definition of the reactions

The user effect is that of a competent user — one who has no problems with the rest of the task, but is likely to get into trouble with the unselected windows.

The definition of the user effect probability functions are shown in figure 9-4. The important two figures are the probability of invoking *work* after *finish* (0.4) and the probability of invoking *move* after *finish* (0.6) — these express the probability of the user making a mistake with selecting the window and we have assumed that the user will make the mistake 2 in 5 times. There are two functions t and t' that take care of the task performance side of the specification. The probability of doing more *work* reactions is defined by t and the probability of invoking a *finish* is defined by t'. For simplicity we assume that t is 1 for several *work* reactions and then t goes to 0 and t' goes to 1.

We can define a state machine to express this user effect graphically. This is shown in figure 9-5. Each node expresses the most recent reaction invoked and the arcs between the nodes express the next reaction to be invoked and probability of that reaction being invoked. The thick arcs show the reactions we are interested in *i.e.* the reactions with a probability greater

$$
\begin{array}{l}
pu_{work}(rs) \hat{=} \\
\quad rs = \Lambda \Rightarrow 1 \\
\quad rs = (work + finish + move)^* work \Rightarrow t \\
\quad rs = (work + finish + move)^* finish \Rightarrow 0.4 \\
\quad rs = (work + finish + move)^* move \Rightarrow 1
\end{array}
\tag{9.44}
$$

$$
\begin{array}{l}
pu_{finish}(rs) \hat{=} \\
\quad rs = \Lambda \Rightarrow 0 \\
\quad rs = (work + finish + move)^* work \Rightarrow t' \\
\quad rs = (work + finish + move)^* finish \Rightarrow 0 \\
\quad rs = (work + finish + move)^* move \Rightarrow 0
\end{array}
\tag{9.45}
$$

$$
\begin{array}{l}
pu_{move}(rs) \hat{=} \\
\quad rs = \Lambda \Rightarrow 0 \\
\quad rs = (work + finish + move)^* work \Rightarrow 0 \\
\quad rs = (work + finish + move)^* finish \Rightarrow 0.6 \\
\quad rs = (work + finish + move)^* move \Rightarrow 0
\end{array}
\tag{9.46}
$$

Figure 9-4: The definition of the user effect

than zero.

Now we can start to consider an interface effect that together with this user effect defines a system that is consistent with the use requirements. In this example this is quite simple as the only points that are crucial are where the user is likely to make a mistake with the windows. As it stands, if we had a user-interface that placed no biasing on the use then we would have a system that does not make the mistake 60% of the time, we therefore need to specify an interface that is going to bias this figure up to 90%. So we define an interface effect that places more emphasis on the reaction *move* after a *finish*. Such an interface effect is shown in figure 9-6. The probabilities in bold denote the overall biasing of the user and interface effects combined.

Essentially we set all the interface effect probabilities to 1/3 which corresponds to a null interface effect and then we 'tweak' the values of the probabilities for the reactions *work* after

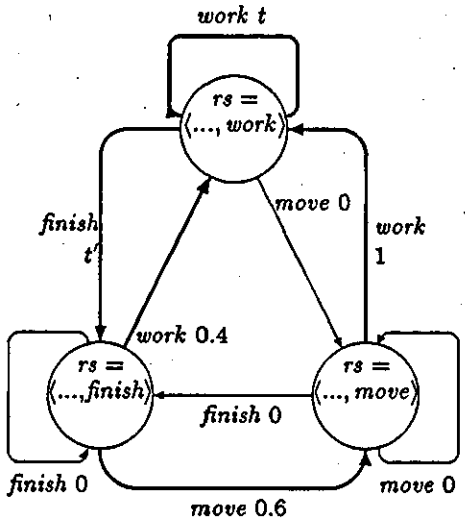


Figure 9-5: The user effect shown graphically

a *finish* and for *move* after *finish* until we get values for the overall biasing that match (approximately) the required 0.1 and 0.9. Because this is the only mistake that we assume our competent user will make (and after all the measurement scheme only captures that mistake) then we assert that the user and interface effects are consistent with the usage distribution.

Some caveats

There are of course several simplifications made here.

Firstly in order to get the interface values of 0.145 and 0.855 we have to correspondingly reduce the interface effect of performing a *finish* after a *finish* to 0. This is acceptable in the case of our competent user who will not do this anyway but if we were considering a less competent user then there is a probability that the user may make this mistake. Whether or not we want to allow the user to is context dependent — using the interface to 'block off' certain parts of legal functionality is not really what the interface is for by our definition. If we *do* want to block out this possibility we should re-specify the underlying functionality to make it illegal rather than using the interface to make it impossible. In making this possibility impossible we also reduce the user freedom in an interaction. There is a trade-off to be made; user freedom against the reduction of user errors. A positive note to be made is that our approach exposes this trade-off

$$
\begin{array}{l}
pi_{work}(rs) \hat{=} \\
\quad rs = \Lambda \qquad\qquad\qquad\qquad\qquad \Rightarrow 1/3 \quad \mathbf{1} \\
\quad rs = (work + finish + move)^* work \Rightarrow 1/3 \quad \mathrm{t} \\
\quad rs = (work + finish + move)^* finish \Rightarrow 0.145 \quad \mathbf{0.102} \\
\quad rs = (work + finish + move)^* move \Rightarrow 1/3 \quad \mathbf{1}
\end{array}
\tag{9.47}
$$

$$
\begin{array}{l}
pi_{finish}(rs) \hat{=} \\
\quad rs = \Lambda \qquad\qquad\qquad\qquad\qquad \Rightarrow 1/3 \quad \mathbf{0} \\
\quad rs = (work + finish + move)^* work \Rightarrow 1/3 \quad \mathrm{t}' \\
\quad rs = (work + finish + move)^* finish \Rightarrow 0 \quad \mathbf{0} \\
\quad rs = (work + finish + move)^* move \Rightarrow 1/3 \quad \mathbf{0}
\end{array}
\tag{9.48}
$$

$$
\begin{array}{l}
pi_{move}(rs) \hat{=} \\
\quad rs = \Lambda \qquad\qquad\qquad\qquad\qquad \Rightarrow 1/3 \quad \mathbf{0} \\
\quad rs = (work + finish + move)^* work \Rightarrow 1/3 \quad \mathbf{0} \\
\quad rs = (work + finish + move)^* finish \Rightarrow 0.855 \quad \mathbf{0.898} \\
\quad rs = (work + finish + move)^* move \Rightarrow 1/3 \quad \mathbf{0}
\end{array}
\tag{9.49}
$$

Figure 9-6: The definition of the interface effect and the resultant overall biasing

clearly to the designer.

Secondly and more subtly we require an interface that increases the probability of performing a *move* after a *finish*, but as it stands the interface effect must be able to discriminate between *finish*es and *work* to be able to do this. If the interface can discriminate between these then there is no reason why the underlying functionality cannot and we should probably use some of the approaches we catalogued when we discussed functional approaches to the problem. A more realistic definition would make *move*s generally more probable whether they follow *finish*es or not. We can do this easily within our approach. But this would considerably complicate the mathematics of calculating the overall biasing and may therefore obscure the fundamentals of the example.

Thirdly the user effect described here only approximately gets the correct value for the overall

biasing. We are satisfied with this approximation because it makes very little sense to define a user effect to an accuracy of four or more decimal places (indeed its debatable that anything more than a couple of decimal places is all that sensible). The user effects are a specification that needs to be passed to an interface designer — the techniques and tools available to the interface designer are most unlikely to be able determine whether interface features have the specified effect to this level of accuracy.

9.3.5 Implementing an interface from the specified user effect

We pass the specified interface to an interface designer to build an interface with that effect. This is where our formal approach finishes — we have produced a formal specification of an interface and it is beyond the scope of this work to provide definitive answers to question of what interface widgets and features produce what biasing.

Let us look at the problem in an informal way though. The most popular interface remedy to the problem that is described in [117] is to make the border of the selected window more prominent and this of course neatly fits in with our interface specification. Making the selected window more prominent increases the probability of the user noticing that they are about to enter data into the wrong window and the more prominent the window the more likely the user is to notice and therefore correctly invoke a *move*. In our example we have assumed a user effect where the mistake is very common and hence we need a very prominent selected window, possibly not just in bright colours but also with a 'fizzy' border that consistently attracts the user's attention and possibly we may also wish to considerably 'grey out' the unselected windows. If the user effect was not so error prone then the interface effect would not be so dramatic and we would not need to resort to such measures.

Our approach also allows for more novel approaches. Usually the selected window is selected and remains selected until something occurs to change that. We may wish to make *move* the 'default' reaction that occurs unless the user actively does something to prevent it — this makes it considerable more likely to occur. Windows might actively have to be 'held' open by the user so that, for example, the user may only be able to enter data into a window when a function key is being held down.

The problem with this interface is that it may interfere with the task itself and reduce the

probability of the user doing the task correctly, the user being more worried about holding function keys down to get the right window to stay open than they are about the task itself. The model as it stands is not expressive enough to capture this. We would have to decompose *work* and *finish* so that we can express whether the task itself is done badly and then we may define a four state measurement scheme consisting of 'task done efficiently and no problems with unselected windows', 'task done efficiently and problems with unselected windows', 'task done badly and no problems with unselected windows' and 'task done badly and problems with unselected windows'. The chances are that *move* being default increases the probability of the task being done badly. In this new model we would have to state a requirement for how often we wish the task to be done efficiently and then make assumptions about the user population and so on.

A compromise in this situation may be to make the main task window default and then make sub-task windows have to be actively held open. This reduces the interference on the main task at least.

9.3.6 Summary of this example

This example has laid out the strategy of our approach to specifying more usable systems...

1. define the functionality of the system,
2. describe how 'good' we want the use of that system to be,
3. make assumptions about what the users are likely to do with the system,
4. specify an interface that is going to bias what the user does so that the use is as good as required, and
5. pass this specification to an interface designer who will implement the actual interface features that match this specification.

As well it has also been shown that the act of formalising a system in an abstract way allows for the designer to consider the system in a clear way. We have been able to split ways of overcoming the unselected windows problem into three categories and discuss them in a fairly elegant way.

1. Functional solutions which rely on the device being able to discriminate when the user has completed the task and then switching window automatically.

2. Interface solutions which make *move* more likely possibly by making the difference between the selected window and the other windows more prominent.

3. Extreme solutions which make *move* more likely but at the cost of interfering with the rest of the task.

These categories were not explicitly captured in [117], we argue that the abstraction level offered by our approach enables such insights to be captured and new perspectives to be analysed.

9.4 Conclusions

We have shown that we can link our approach to the usability properties approach in section 9.1. In chapter 5 we showed how we can define requirements and then develop these requirements into abstract system specifications and suggested how we can map these specifications onto implementations. In section 9.2 we showed a middle ground — how we can add detail to specifications by adding more reactions and more complicated reactions to a specification. Finally we showed in section 9.3 how we can feed psychologically valid information (or at least information that can be exposed to psychological investigation) into the design of user interfaces in order to make the use of a system closer to good use.

We have therefore shown several facets of our work and hopefully shown how they interrelate in order to produce an integrated approach for designing and building better interactive systems. We contend also that our approach can comfortably deal with the three levels of abstraction detailed by Dearden and Harrison [35]. Dearden and Harrison argue that PIE models are so abstract that it is difficult to express genuinely useful interaction concerns. Indeed the numerous caveats that Dix suggests for the properties demonstrate this point. The problems we showed for formulating reachability show that our definition of requirements will be even more abstract than a PIE model, hence Dearden and Harrison's arguments against excessive abstraction apply even more so to our statements of requirements. Furthermore Dearden and Harrison question approaches that formalise the interface (such as the interactors model) in that

their representations are too concrete to be re-used across several applications. Dearden and Harrison therefore suggest a middle abstraction ground where generic models of different types of system are described.

We assert that our approach is flexible enough to cover all three of these levels of abstraction. The nuclear power plant example we used in section 5.6.4 to show the benefit of using a reaction style specification is an example of a specification expressed at this middle level of abstraction. The specification expresses a central representation (in this case the valve icon) which is manipulated by two agents. The specification describes how the central representation changes according to stimuli from the two agents. The specification shows a marked similarity to that presented in [23] which describes how a scroll-bar representation reacts to manipulations from the scroll buttons being pressed and to the movement of the window in a windowed data structure.

Both these examples can be thought of as cases of 'direct manipulation' systems where there is layer of representation between the user and the functionality. The user can change the representation and expect that the state of the underlying functionality to change in concert with it. In a similar way the underlying functionality can have its state changed (possibly by stimuli not generated by the user) and must reflect those changes onto the representation so the user can be aware of them.

Although we contend that our framework is general enough to capture models at all three of Dearden and Harrison's levels we question their assertion that one of these levels is in some way 'better' than the others. Properties and analyses tend to have a level of abstraction at which they are best expressed and because an interactive system has many relevant properties that need capturing then it is likely that we need an approach that can span many levels of abstraction and has well defined mapping between them.

Chapter 10

Summary, conclusions and agenda for future work

To summarise the thesis we first compare what has been reported to the aims and objectives laid down in chapter 2. We then discuss the use of the framework in a generalised context, suggesting how it should guide both functional and non-functional requirements for systems. We discuss how the repeated use of the framework may 'firm up' HCI theory. We then look at the weaknesses of the work and propose an agenda for further work and conclude the thesis.

10.1 Comparing the work done to the aims and objectives

10.1.1 Interactive systems as reactive systems

We have proposed a technique for specifying reactive systems and then specialised it to a technique for specifying interactive systems. Broadly what distinguishes the two is that we have added probablistic apparatus for describing the *use* of interactive systems.

10.1.2 Including usability as part of the system synthesis process

We have contended that a usable system is one the use of which is similar to that of an interactive system where the user population is expert. We have augmented the design process with describing what the behaviour of such an expert user population would be and describing

measurement schemes that express how 'near' to this expert behaviour some activity is. We can then express requirements for the proposed use in terms of how likely it is that the use obtains a certain value according to the measurement scheme. We then showed how we can specify user-interfaces in terms of the biasing they exert on use and how we can show consistency between a specified interface and the required use.

Therefore we can require different levels of usability and guide the design of user-interfaces such that the use of a system matches those requirements.

We are aware, however, that the theory needed to fully guide design decisions made within our framework is weak. Although our main objective was the definition of a framework that aids in the *construction* of interactive systems we will argue in the next section that the framework can be used in a more analytic manner to develop useful HCI theory.

10.1.3 Notations

We have purposely treated our notations lightly. The notations we have presented were simply those that we felt comfortable with. The author was introduced to formal methods with VDM and hence the notations (particularly for computations) have a VDM feel to them. We do not claim this notation to be any better or worse than any other notation although we believe them to be adequate for what we have presented here. It is frequently claimed (*e.g.* [3]) that non-formal workers prefer to work with graphical notations. There is little empirical justification for this claim other than [67] where it was shown that specifiers are happier with natural language descriptions but (possibly paradoxically) made less errors with a temporal logic. There exist graphical notations [75, 59] that could be fairly easily integrated into our approach. We discuss these in more detail in the next section.

10.1.4 Viewing usability 'from above'

We have contrasted our approach to usability to the usability properties approach in section 7.7. In section 9.1 we captured some of the 'classic' usability properties in temporal logic. A more extensive collection of usability properties expressed in temporal logic is shown in [83] (although branching temporal action logic is used). We contend that our approach to usability is more holistic and encompassing than the usability properties approach.

10.1.5 Towards statistical and probabilistic models

We have proposed ways of incorporating probabilistic information into models of interactive systems. We have kept the semantics underlying the probabilistic side deliberately very simple as more complex models require considerably more complex semantic underpinnings [68]. We note that our models may share much in common with Markov models and that an investigation of such models may be warranted to show the tractability (or otherwise) of the mathematics required to make our approach work.

10.1.6 A comparative framework

The compilation of a collection of case studies within our framework has been judged to be beyond the scope of this work. Without a collection of case studies which is generally accepted to be an adequate coverage of the field then the work involved in attempting to compile such a collection would be considerable, never mind expressing all the examples in our framework. We have shown examples that cover a wide range of generality, including the HCI 'white rat', the word processor.

10.2 Developing interactive systems

Assume we are developing an interactive system.

10.2.1 Developing functionality

First of all we would determine requirements for the system based on what task the customer wants the system to perform. Note that the customer need not be the end-users and we consider the system to include the users. This requirements gathering is likely to be informal and based on questions asked of the customer and analysis of existing systems.

We can now formalise the requirements by describing the functionality required of the system. We capture the functionality by describing the state space that is shared by the proposed device and the users. This shared state space can typically consist of the keyboard, mouse and screen *etc.* but will be expressed at a level of abstraction that does not dictate actual hardware.

The formal requirements express the relationships that hold between the various entities in the shared space. A specification is then developed from requirements which describes in more detail the invocations made of the device and the responses generated by the device. We pair the invocations and responses together into reactions.

We can decompose the reactions into sub-reactions using hierarchical task analysis methods. A task analysis suggests a collection of tools that the user will find useful in the achievement of a given task. Reactions should correspond closely to to tools recommended by a task analysis. A hierarchical task analysis suggests how a tool can be iteratively decomposed into sub-tools. For example an 'edit-text' tool would be decomposed into 'insert-char', 'delete', 'cut', 'copy', 'paste' *etc.* A hierarchical task analysis therefore should relate closely to the decomposition of reactions.

We then refine the responses by describing how we wish the device to generate the responses and how we want the user to make requests.

Functional decomposition, task analysis and refinement are well documented and nothing we have described so far in this section should be surprising.

10.2.2 Developing the (non-functional) usage requirements

We claim that our approach is novel in how it deals with non-functional requirements, in particular how we expect the system to perform the tasks collected in the requirements and detailed in the task analyses. We argue that for any task there is an optimal manner in which the user can use the tools given to achieve that task and we further argue that a usable system is one which makes this optimal behaviour more likely.

Usability requirements

The achievement of a task is modelled as a sequence of reactions which places the system in some goal state. We can consider a collection of those sequences to be optimal and we can also devise a measurement scheme that takes an arbitrary reaction sequence and determines a measure of how close that sequence is to the optimal.

The definition of optimality and measurement schemes is heavily context dependent and relies on the judgement of the developer. Simplistically we could consider the shortest sequence

of reactions that results in a goal state to be the optimal. Furthermore the measurement scheme is proportional to the length of a reaction sequence (the shorter the better). Such an approach would be unrealistic for all but the simplest tasks however, a more realistic approach would take into account a plethora of other issues.

Given a way of capturing how 'good' a reaction sequence the developer can describe the usability requirements. This can be done in terms of matching goodness of reaction sequence to the probability of a reaction sequence of that goodness occurring. A precise set of requirements can be described using a graph plotting goodness against probability. Less precise descriptions of the form '50% of the time the system performs adequately, 20% of the time excellently, 20% of the time badly and 10% of the time dreadfully' are also perfectly acceptable. Our framework asserts no level of precision. Hence a precise graph based on a very detailed measurement scheme maps into exactly the same mathematical entity as less precise textual requirements.

Levels of abstraction and modularity

At higher levels of abstraction the generality of tasks may mean that little useful can be described for usability requirements. A word processing task (producing a document) is a very general thing. Even if it is possible to define optimality and a measurement scheme it would be debatable whether a developer would wish to assert that the system be heavily biased to the optimal — one of the benefits of interactive systems is that they leave plenty of room for the user to explore. User freedom must be traded against biasing to the optimal.

We argue, however, that at lower levels of abstraction it is possible to determine optimality in a quite precise manner. Consider a dialogue box requesting that the user select a file for opening. We can be definite about good and bad ways of interacting with the dialogue box in much more precise way than we can be about good and bad ways of interacting with a word processor. Recall the discussion above where we suggested that reactions equate to tools for performing tasks — we suggest that the more decomposed the reactions, and hence the lower the abstraction of the task then the more precise we can be about optimal ways of using those reactions to perform that task. We can argue, however, that a 'good' reaction sequence expressed at a high level of abstraction is built from good sub-reacton sequences. It may not be possible to be explicit about a good use of a word processor, but we are likely to be explicit about good

uses of the tools supplied by the word processor.

Usability requirements inherit the modularity of the reactions defined in the development of the functionality. For each level of decomposition optimality can be defined for the use of each collection of reactions. A question not addressed in this work is how genuinely modular usability requirements can be. We can define optimality for an open file dialogue box, but how independent of the rest of the system is this optimality? The arguments of Carroll [28] we outlined in chapter 3 suggest that usability ideas are not amenable to simple modularity in the same way that functionality is. Further investigation is required.

Specifying user interfaces

Given usability requirements we have suggested that we can specify user interfaces and make assumptions about the user population such that the system fulfills the usability requirements.

The derivation of user effects is problematic. We argue that we can still define interfaces without an explicit user effect — if we assume the user population to be a random variable then an interface that biases a random user population to the optimal will bias any population to the optimal. We suggested that the amount of biasing caused by an interface can be thought of as the 'strength' of the interface. In most cases capturing user effects allows the developer to define weaker interfaces that still fulfill the usability requirements.

A developer can use rapid prototyping techniques to capture user and interface effects. Given a functional specification of a system the developer can rapidly prototype an interface for that functionality and test it on a user population representative of the proposed user population. By keeping the user population constant and varying the prototyped interfaces the developer can capture user effects — variations in the use will be caused by the various interfaces and normalising for the interfaces will determine the user effect. Obviously there will be confounding effects that need controlling for, for example learning effects.

10.2.3 Building HCI theory

Repeatedly developing interactive systems in this way will allow for developers to collect together user and interface effects and to refine already defined effects. In this way the framework is being used in a more analytic manner. The first few times the framework is used by a developer will be

very experimental and there will be little guarantee of success (at any rate no more guarantee of success than if the system was constructed in the more usual *ad hoc* manner). However because systems are being constructed within the framework the design decisions made are explicit and the results of those decisions can be captured in a explicit way.

Hence HCI theory is captured and can be reused in other interactive system development and can be exposed to critical analysis by human factors workers. Essentially the repeated use of the framework takes the heuristic knowledge of HCI workers and captures it formally and explicitly.

Ideally HCI theory which has been exposed by the use of our framework should be captured in such a way that it can be reused not only in our framework but in other HCI development as well. In a similar way repeatedly using our framework should not be about reinventing wheels — HCI theory that has already been captured in other work should be able to be reused within our framework. Such claims of transfer are well beyond the scope of this thesis. However the work of [13] shows how several formal HCI approaches can be harnessed into a single design space and that there is transfer between them.

10.3 An agenda for further work

As the framework we have presented is rather broad in scope there is plenty of room for further work to make our approach more practical, to make the semantics more expressive and to ground it more firmly in psychological theory. Some of the avenues that we feel would be beneficial to follow are listed in this section. There are likely to be several others.

10.3.1 HCI practitioner-friendly notations

We have repeatedly expressed worries about the significance attached to the choice of notations. However it may be useful to move the notations we have used into a more graphical domain for the benefit of workers who are used to such notations.

A particularly interesting approach to graphical notations is presented by Lamport [75] called 'TLA in pictures'. Lamport attempts to overcome the problems of complexity in graphical specifications by proposing a graphical notation that need not express *all* of a specification, only

salient parts of it. A full formal treatment of the notation is given and it can therefore be easily shown that if D is a diagram and *spec* is a specification then D is a representation of part of that specification if $spec \Rightarrow D$. Hence it is possible to not only show several diagrams of salient parts of a specification, we can also use different diagrams to give complementary views of the same part of a specification.

TLA in pictures is therefore not a complete specification technique (particularly as they only express safety properties), it is an aid to the comprehension of TLA specifications.

We feel that we could propose a similar notations for RSSL and ISSL, in fact in section 9.3 we used a graphical state machine to extract the user effect from a specification is exactly the spirit of TLA in pictures. This was only a rough sketch though and we need to show a formal semantics for the notation for it to be a genuinely useful tool. In particular RSSL specifications split computations into obligatory and optional computations and guarantee liveness for the obligatory computations. TLA in pictures cannot express liveness and therefore we would need to carefully consider how we were to express obligation graphically if at all possible. We would also like to integrate the work on state-charts [59] which allows modularity in graphical notations.

10.3.2 Approximation and tolerances

At many points in this work we have alluded to the need for approximation in the production of models. We are aware that psychological HCI can in many cases not provide precise results and therefore it makes no sense to insist that its results are straight-jacketed into a formal framework.

An interesting and practical avenue would be to redefine the work on uses and ISSL specifications allowing for approximations within tolerated boundaries. In section 9.3 we worked through an example concentrating on the use of a system, the result being a specification for an interface that *approximately* generates use that is consistent with the requirements. It would be valuable to be able to require some use from a system and also describe a tolerance for that requirement — how close to the required use an implemented use can be and still considered to be acceptable.

It is worth reiterating that the evaluation of models within our framework can only be as precise as the description of the entities that we have used in the framework. It is no good demanding a high level of accuracy in the use displayed by an implemented system if

the specification of that use is based on an approximate model of user effect and there is little evidence that the actual interface features implemented have the interface effect that has been specified for them. Another source of approximation is the measurement scheme used to capture how good an interaction is. An interesting question is how these three sources of approximation combine — if x is the approximation due to the measurement scheme, y the approximation due to the user effect and z the approximation due to implementation of the interface effect then is the overall approximation in the model $x+y+z$, $x \times y \times z$ or c^{x+y+z} (where c is some constant)? If we can get even a rough answer to this question then we can dictate how much emphasis should be placed on the result of modelling a system using our approach.

10.3.3 True concurrency

The semantic definition for RSSL specifications is based on a 'pseudo-interleaved' concurrent model. We have been careful to define a specification language that separates out all the points where concurrently performing computations may interfere with one another (*i.e.* the read and write phases). The semantics describe these phases as being interleaved and the rest of the functionality as possibly occurring concurrently.

Hence the semantics describe a 'canonical' model of activity, the idea being that we can implement a system that has the 'same effect' as these canonical activities. We have not described in any detail how we can map this canonical model to a genuinely concurrent model where the read and write phases can occur concurrently — we need some way of formalising what 'same effect' means.

We might do this by instigating a thorough redefinition of the RSSL semantics that relate *partial orderings* of computations to activities rather than *sequences* of computations as we have done in this work.

Alternatively we may take an activity (which is a total ordering) of state changes and apply the following rule; 'Assume we can describe an activity as a sequence of state changes denoted $\langle A; B; C \ldots \rangle$ where A, B and C are state changes. If there are two valid activities α_1 and α_2 which are the same except that...

$$\alpha_1 = \langle \ldots A; B \ldots \rangle$$

$$\alpha_2 = \langle \ldots B; A \ldots \rangle$$

...and the state change caused by $A;B$ in α_1 is the same as caused by $B;A$ in α_2 then we can implement a system where it does not matter which order A and B occur in — *i.e.* we can implement them concurrently.'

Unfortunately things are not that simple. Consider the following counter example — two computations A and B increment and decrement the value of variable x respectively. Each computation copies the value of x to some internal store, increments or decrements the value and copies it back to x. If the two are performed in sequence then it does not matter which is performed first, the net result is no change to x. However once we put them in parallel then the two may copy x to their internal store increment and decrement those copies concurrently and then copy them back to x. The net effect in this case is either an increment and decrement depending on which computation completes last. Hence we need to consider a rule like the one given above but with several constraining clauses.

10.3.4 Obligation

The notion of obligation we have included in the RSSL semantics is very weak — surprisingly so. When an obligatory computation is enabled the system must do something, but the something it does need not be the launching of that obligatory computation. Hence there are situations where an obligatory computation may become enabled and something may happen before that computation has chance to be launched to disable it once more. However an obligatory computation cannot be indefinitely locked out in this manner — fairness ensures this.

We have allowed this very weak notion of obligation on purpose, because otherwise an indefinitely enabled obligatory computation would lock out everything else. Consider a model where the environment enables a kernel computation which repeatedly performs its processing until the environment actively does something to disable it. Under a strong notion of obligation (where if an obligated computation is enabled then it is possible and nothing else is) then once the environment has enabled the kernel computation it would never be able to stop it, because everything apart from the kernel computation would be impossible.

So we have a choice between a very strong notion of obligation that easily traps the unwary specifier, and a fairly weak notion that may result in a surprising amount of inactivity for

the kernel. Our problem is that the heuristic definition of obligation; 'the system is obliged to perform enabled kernel computations *if processing resources are available*' is very difficult to sensibly formalise because the semantics for RSSL specifications deliberately abstract away from any notion of the machines that we may wish to implement the described systems. Hence the notion of 'processing resources' being available is tricky to capture, there being no explicit model of processing resources.

The legal behaviour for an RSSL specification should cover all possible machine configurations, from single processor to distributed multi-processor machines. In section 5.6.2 we showed how we can explicitly limit the number of concurrently occurring computations by putting extra conditions on the enabling clauses, so that computations cannot be launched if there are not enough processing resources available. However there is no way of ensuring that if (say) there are ten processors available and there are ten obligatory computations enabled then all those computations will be processed concurrently. The semantics presented in chapter 6 ensure that at least one of the computations will get processed, but do not ensure that any more than one get processed, which may seem surprising.

We could propose a new more operational semantics for RSSL where the processing resources are included as a special variable (in the same way that time is considered to be a special variable). Therefore we could capture a notion of obligation that is rather closer to the heuristic definition of obligation, though at the cost of some abstraction.

10.3.5 Proof techniques and methodologies

We have assumed that we can import the proof technique of TLA into RSSL because we contend that RSSL is a specialisation TLA, not an extension to it (see the arguments in [23]). To be fully confident of this claim we would need to formally prove it.

Even if we do prove this then we are aware that the proof techniques presented for TLA are *only* techniques, they are not a full methodology, in the sense that TLA asserts that we can prove things about TLA specifications, but gives only very sparse information about how to actually perform those proofs. As TLA and similar notations become more widely used and investigated then a proof methodology should emerge as well as automated tools to aid the methodology.

10.3.6 User models

In order for our approach to produce more predictive results it needs to be interfaced to a good user modelling technique. We argue that we can still obtain some level of useful results by simply assuming the user is a random variable — if we can design an interface that is 'strong' enough to bias totally random users into a good use then it should also work with actual users. However considerably more leverage would be obtained if a sensible user model was used.

Many of the user models presented for evaluating HCI systems are deterministic and therefore we may have problems linking them to our probabilistic approach. However a little imagination should overcome this.

Consider the PUM [118] approach based on problem solving techniques. The user model is given a model of the device and some knowledge about that device and some knowledge of the goals. PUM predicts how the user will use this information to make a rational choice of what to do next. Typically there will be several such rational choices and therefore the analyst can employ several other heuristics and guidelines to decide which choice is the most likely for the user to make. PUM uses these heuristics in order to constrain the behaviour space that is to be analysed. We can use these heuristics to encode a measure of the probability of what the user does next instead of using them to constrain the behaviour space. If a certain activity contradicts several heuristics then we can predict that it is unlikely whereas if several heuristics promote an activity then we can predict it as being likely. Again the ideas of approximation come into play here and we also need to consider the psychological justifications for any heuristics.

This is a thumb-nail sketch of how we might interface a PUM model to our approach and presumably similar arguments may be made for interfacing other user models to our approach. Another avenue to investigate is the definition of a user model specifically targeted at our approach so that ideas of probability are addressed directly.

Another approach that does away with user models altogether is the collection of behaviour data in a 'black-box' way. We might observe users interacting with a certain device, then change the device slightly and infer that changes in the use are due to the change in the device. In this way we can build up a picture of user effect in a behaviourist way rather than by using a cognitive user model that attempts to explain the user effect. Suggesting a return to behaviourist psychology is likely to be contentious and there would be an awful lot of work involved in building

up a useful picture of a user effect in this way. It is debatable whether that work would pay off in terms of the improvements to interactive systems it would supply.

10.3.7 Towards an engineering approach

We have pointed out several places in this work where the application of our framework allows for explicit reasoning behind engineering decisions. For example in chapter 7 we suggested that the biasing exerted by a user interface is likely to be proportional to the cost of producing that interface. Therefore we can trade-off the benefit of improving user performance with a better interface against the cost of producing that interface.

Another important consideration when moving our approach into a more practical domain is that it should 'hide' the underlying theory whilst applying the results of that theory to the design process. A software engineer should not need to know the full psychological reasons behind a user effect, but should be able to use that user effect in the design of user interfaces.

An ultimate aim of this work would be to provide a library of user effects expressed for a variety of tasks and classes of users, a similar library of interface widgets and features with defined interface effects. These libraries should be available to engineers to apply in a design setting and also available to human factors workers to justify and explain why the effects relate to the interface features and so on. In this way such libraries would constitute a valuable, practical and explicitly justified tool in interactive system design.

10.4 Conclusions

We presented a framework of ideas for capturing psychologically relevant ideas and feeding them into the design of interactive systems. We have presented this framework in a formal manner so that the ideas and concepts captured can, ultimately, be rigorously demonstrated to hold in implemented systems. Having done so, we also contend that our framework is useful in a more informal context — there is much anecdotal evidence from the Amodeus project that simply the act of abstractly describing an interactive system can expose many previously hidden errors and inconsistencies. Furthermore we can argue that the act of taking psychological evidence and expressing it within the mathematical structures we have presented forces us to justify and

rationalise this psychological evidence. In traditional HCI circles 'psychological evidence' is often captured as heuristic knowledge and craft skill of usability experts. It was argued in chapter 3 that such evidence does not generalise and cannot easily be reused unless it is expressed in the form of a theory. We contend that the use of our framework not only feeds existing theories into the interactive system design process, but also helps in codifying heuristic and craft evidence in a more formal and theoretical manner.

However, in describing a framework we have necessarily made assumptions about what we consider a desirable interactive system to be (and not be). We therefore must accept the fact that the framework will bias the design of interactive systems. There are arguments for and against such an approach — there is a loss of generality which could be argued to be undesirable, yet a framework that is fully general is not a framework at all. We *have* made the assumptions about the framework explicit as it was described, however it may not be reasonable to expect a developer intending to use our framework to be fully aware of these assumptions. A similar argument can be raised in relation to the concentration on notations found in some other formal HCI work — our worry is that notations step away from the underlying semantics and may therefore hide various assumptions. Both our framework and notations hide assumptions to a certain extent.

We do not intend for our framework to be a formal 'straight jacket' on the field of HCI however. We are aware that there are many aspects of usability and interface design that cannot be captured formally in a sensible manner. The mapping from an interface specification expressed as an interface effect to actual interface features is very loose and vague, indeed resting on the skill and heuristic knowledge of usability experts. What this thesis suggested is a process whereby the reasoning that leads the design process to the interface effect is made explicit and therefore open to scrutiny. In a nutshell we aim to explain *what* an interface does and *why* we want the interface to do it. *How* the interface does it is where our work finishes. Experience with our framework will firm up the mapping from interface effect to interface features — if an interface implementer repeatedly finds that he implements the same interface effect with the same interface features then we contend that this is a 'fact' (in Storrs' [109] terminology) to be added to HCI theory. The proposition of user effects and measurement schemes are other points in our framework with similarly weak mappings from psychological and usability evidence to our

structures.

We have furthermore proposed a link from the probabilistic models used to describe psychological information to the discrete models used to describe device functionality. We have suggested refinement strategies and techniques for taking abstract discrete models towards concrete implementations. Again these refinement techniques are not as formally rigourous as would ideally be hoped. We understand that there is work underway by researchers in the TLA group to pin down the methodology of refinement of TLA and similar specifications [76] and we contend that we can tap our framework into this work as it progresses.

All in all our framework is broad in scope and therefore asks many more questions than it answers. Crucially though, it provides a sensible well defined context which answers can be fed back into once they are captured. Furthermore it provides a structure for better, more abstract and sensible asking of questions and a way of making the reasoning behind the need for the asking of those questions to be made explicit.

Bibliography

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(2):366–393, 1995.

[2] G. Abowd. *Formal Aspects of Human-Computer Interaction.* PhD thesis, University of Oxford, 1991.

[3] G. Abowd, A Dix, and M. Harrison. State of the art: Formal aspects of user interfaces. Internal report, HCI Group, Dept. of Comp. Sci., University of York, 1990.

[4] G. D. Abowd, J. Coutaz, and L. Nigay. Structuring the space of interactive system properties. In J. Larson and C. Unger, editors, *Engineering for Human-Computer Interaction*, pages 113–129. Elsevier Science Publishers, 1992.

[5] G. D. Abowd and A. J. Dix. Integrating status and event phenomena in formal specifications of interactive systems. *Softwate Engineering Notices*, 19(5):44–52, 1994.

[6] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[7] R. J. R. Back and R. Kurki-Suounio. Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[8] P. Barnard. Interacting cognitive subsytems: A psycholinguistic approach to short term memory. In *Progress in the psychology of language*, volume 2, pages 197–258. Lawrence Erlbaum, 1985.

[9] P. Barnard. Bridging between basic theroies and the artifacts of human-computer interaction. In *Designing Interaction: Psychology at the human-computer interface*, Cambridge Series on Human-Computer Interaction, pages 103–127. Cambridge University Press, 1991.

[10] P. Barnard and M. D. Harrison. Integrating cognitive and system models in human computer interaction. In A. G. Sutcliffe and L. A. Macauley, editors, *People and Computers V*. Cambridge University Press, 1989.

[11] P. Barnard and J. May. Cognitive modelling for user requirements. In P. F. Byerley, Barnard P. J., and J. May, editors, *Computers, Communications and Usability: Design issues, research and methods for integrated service*, pages 101–145. Elsevier, 1993.

[12] L. Bass. Working group on formal methods in HCI and software engineering. In R. N. Taylor and C. Coutaz, editors, *Software engineering and human-computer interaction (Lecture notes in computer science vol. 896)*, pages 14–16. Springer Verlag, 1995.

[13] V. Bellotti, A. Blandford, D. Duke, A. MacLean, J. May, and L. Nigay. Interpersonal access control in computer mediated communications: A systematic analysis of the design space. *Human Computer Interaction*, 11(4):357–432, 1996.

[14] A. Blandford, R. J. Butterworth, and J. P. Good. Users as rational interacting agents: formalising assumptions about cognition and interaction. In M. D. Harrison and J. C. Torres, editors, *Proceedings of Design, Specification and Verification of Interactive Systems '97*, pages 51–66. Springer, 1997. Page numbers refer to pre-print copy.

[15] A. Blandford and R. M. Young. Specifying user knowledge for the design of interactive systems. *Software Engineering Journal*, pages 323–333, 1996.

[16] A. E. Blandford, M. D. Harrison, and P. J. Barnard. Integrating user requirements and system specification. In P. F. Byerley, P. J. Barnard, and J. May, editors, *Computers, Communication and Usability: Design Issues, Research and Methods for Integrated Services*, pages 165–196. Elsevier, 1993.

[17] A. E. Blandford, M. D. Harrison, and P. J. Barnard. Using Interaction Framework to guide the design of interactive systems. *International journal of human-computer studies*, 43:101–130, 1995.

[18] A. E. Blandford and R. M. Young. Developing runnable user models: Separating the problem solving techniques from the domain knowledge. In J. Alty, D. Diaper, and S. Guest, editors, *People and Computers VIII, Proceedings of HCI'93*, pages 111–122. Cambridge University Press, 1993.

[19] P. A. Booth. Redefining software: a comment on Thimbleby's paper. *Interacting with computers*, 2(1):26–32, 1990.

[20] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *Computer*, pages 56–63, April 1995.

[21] P. Brun and M. Beaudouin-Lafon. A taxonomy and evaluation of formalisms for the specification of intecative systems. In *People and Computers X*, pages 197–212, 1995.

[22] 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, *Proceedings of Design, Specification and Verification of Interactive Systems '95*, pages 174–192. Springer, 1995.

[23] R. J. Butterworth and D. J. Cooke. Using temporal logic in the specification of reactive and interactive systems. In *Formal Aspects of the Human Computer Interface.* BCS-FACS, Springer Verlag, 1996.

[24] R. J. Butterworth and D. J. Cooke. On biasing behaviour to the optimal. In M. D. Harrison and J. C. Torres, editors, *Proceedings of Design, Specification and Verification of Interactive Systems '97*, pages 323–340. Springer, 1997. Page numbers refer to pre-print copy.

[25] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Assoc, 1983.

[26] J. M. Carroll. Taking artifacts seriously. In S. Maas and H. Oberquelle, editors, *Software-Ergonomie '89*, pages 36–50, 1989.

[27] J. M. Carroll. Infinite detail and emulation in an ontologically minimized HCI. In J. C. Chew and J. Whiteside, editors, *Empowering People, Proceedings of CHI'90 Conference*, pages 321–327, 1990.

[28] J. M. Carroll, editor. *Designing Interaction: Psychology at the human-computer interface.* Cambridge Series on Human-Computer Interaction. Cambridge University Press, 1991.

[29] J. M. Carroll and R. L. Campbell. Softening up hard science: Reply to Newell and Card. *Human-Computer Interaction*, 2:227–249, 1986.

[30] J. M. Carroll and R. L. Campbell. Artifacts as psychological theories: The case of human-computer interaction. *Behaviour and Information Technology*, 8:247–256, 1989.

[31] J. M. Carroll, W. A. Kellog, and M. B. Rosson. The task-artifact cycle. In *Designing Interaction: Psychology at the human-computer interface*, Cambridge Series on Human-Computer Interaction, pages 74–102. Cambridge University Press, 1991.

[32] G. Cockton. Where do we draw the line? Derivation and evaluation of user interface software separation rules. In M. Harrison and A. F. Monk, editors, *People and Computers: Designing for Usability*, pages 417–432. Cambridge University Press, 1986.

[33] J. Coutaz. PAC, an object oriented model for dialogue design. In H. J. Bullinger and B. Shackel, editors, *Human-computer interaction — INTERACT'87*, pages 431–436. North Holland, 1987.

[34] B. De Carolis and F. De Rosis. Modelling adaptive interaction of OPADE by Petri nets. *SIGCHI Bulletin*, 26(2):48–52, 1994.

[35] A. M. Dearden and M. D. Harrison. Abstract models for HCI. *International journal of human-computer studies*, 46:151–177, 1997.

[36] E. W. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

[37] A. Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, pages 334–346, November 1996.

[38] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-computer interaction.* Prentice Hall, 1993.

[39] A. J. Dix. The myth of the infinitely fast machine. In D. Diaper and R. Winder, editors, *People and Computers III, HCI'87*, pages 215–228. Cambridge University Press, 1987.

[40] A. J. Dix. Nondeterminism as a paradigm for understanding the user interface. In *Formal Methods in Human-Computer Interaction*, Cambridge series on HCI, pages 97–127. Cambridge Uni. Press, 1990.

[41] A. J. Dix. *Formal Methods for Interactive Systems.* Computers and People Series. Academic Press, 1991.

[42] A. J. Dix and M. Harrison. Principles and interaction models for window managers. In M. Harrison and A. F. Monk, editors, *People and Computers: Designing for Usability*, pages 352–366. Cambridge University Press, 1986.

[43] A. J. Dix and M. D. Harrison. Interactive systems design and formal development are incompatible. In J. A. McDermid, editor, *Proceedings 1988 Refinement Workshop.* Butterworth Scientific, 1989.

[44] A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *People and Computers: Designing the Interface*, pages 13–22. Cambridge University Press, 1985.

[45] J. Dowell and J. Long. Towards a conception for an engineering discipline of human factors. *Ergonomics*, 32(11):1513–1535, 1989.

[46] D. Duce and D. Duke. The formalisation of a cognitive architecture and its application to reasoning about human computer interaction. Submitted to FACS for publication, 1996.

[47] D. J. Duke, P. J. Barnard, D. A. Duce, and May J. Syndetic modelling. Submitted for journal publication, 1996.

[48] D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.

[49] D. J. Duke and M. D. Harrison. Connections from A(V) to Z. Technical Report SM/WP21, Amodeus project. ESPRIT BRA 7040, 1994.

[50] D. J. Duke and M. D. Harrison. FSM: Overview and worked examples. Technical Report SM/WP44, Amodeus ESPRIT BRA 7040, 1994.

[51] D. J. Duke and M. D. Harrison. Event model of human-system interaction. *Software Engineering Journal*, 10(1):3–12, 1995.

[52] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In *Technology of Object-Oriented Languages and Systems: TOOLS*. Prentice Hall, 1991.

[53] E. Elwert and E. Schlungbaum. Modelling and generation of graphical user interfaces in the TADEUS approach. In P. Palanque and R. Bastide, editors, *Proceedings of Design, Specification and Verification of Interactive Systems '95*, pages 193–208. Springer, 1995.

[54] G. Faconti and F. Paternó. An approach to the formal specification of the components of an interaction. In C. Vandoni and D. Duce, editors, *Eurographics 90*, pages 481–494. North-Holland, 1990.

[55] N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[56] P. D. Gray, D. England, and S. McGowan. XUAN: Enhancing UAN to capture temporal relations among actions. In G. Cockton, S. W. Draper, and G. R. S. Weir, editors, *People and Computers IX*, pages 301–312. Cambridge University Press, 1994.

[57] P. D. Gray and C. Johnson. Requirements for the next generation of user interface specification languages. In P. Palanque and R. Bastide, editors, *Proceedings of Design, Specification and Verification of Interactive Systems '95*, pages 113–133. Springer, 1995.

[58] J. Grudin. The case against user interface consistency. *Communications of the ACM*, 32(10):1164–1173, 1989.

[59] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[60] M. Harrison and A. J. Dix. A state model of direct manipulation in interactive systems. In *Formal Methods in Human-Computer Interaction*, Cambridge series on HCI, pages 129–151. Cambridge Uni. Press, 1990.

[61] M. D. Harrison and D. J. Duke. A review of formalisms for describing interactive behaviour. In R. N. Taylor and C. Coutaz, editors, *Software engineering and human-computer interaction (Lecture notes in computer science vol. 896)*, pages 49–75. Springer Verlag, 1995.

[62] M. D. Harrison, C. R. Roast, and P. C. Wright. Complementary methods for the iterative design of interactive systems. In *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, pages 651–658. Elsevier Scientific, 1989.

[63] D. Hix and H. R. Hartson. *Developing User Interfaces: ensuring usability through product and process.* J. Wiley, 1993.

[64] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[65] A. Howes and S. J. Payne. Display-based competence: towards user-models for menu driven interfaces. *International Journal of Man-Machine Studies*, 33:637–655, 1990.

[66] R. J. K Jacob. Using formal specifications in the design of a human-computer interface. *Communications of the ACM*, 26(4):259–264, 1983.

[67] C. Johnson. The evaluation of user interface notations. In F. Bodart and J. Vanderdonckt, editors, *Proceedings of Design, Specification and Verification of Interactive Systems '96*, pages 188–206. Springer, 1996.

[68] C. Jones. *Probablistic Non-determinism.* PhD thesis, University of Edinburgh, 1990.

[69] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Intomation Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP, North-Holland, 1983.

[70] C. B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice Hall International, 1986.

[71] D. E. Kieras and D. E. Meyer. An overview if the EPIC architecture for cognition and performance with application to human-computer interaction. Technical Report TR-95/ONR-EPIC-5, EPIC, 1995.

[72] D. E Kieras and P. G. Polson. An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*, 22:365–394, 1985.

[73] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions in Software Engineering*, 3:125–143, 1977.

[74] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[75] L. Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, 1995.

[76] L. Lamport. Personal communication — 'Re: A TLA related question...'. email meassage id: 9608051619.AA05596@hemlock.pa.dec.com, 1996.

[77] T. K. Landauer. Lets get real: A position paper on the role of cognitive psychology in the design of humanly useful and usuable systems. In *Designing Interaction: Psychology at the human-computer interface*, Cambridge Series on Human-Computer Interaction, pages 60–73. Cambridge University Press, 1991.

[78] B. Laurel. *Computers as theatre.* Addison Wesley, second edition, 1993.

[79] J. Long. Theory in Human-Computer Interaction. In *IEE Colloquim on 'Theory in Human-Computer Interaction (HCI)' (Digest No.192).* IEE, 1991.

[80] J. B. Long. Cognitive ergonomics and human-computer interaction: An introduction. In *Cognitive Ergonomics and Human Computer Interaction*, pages 4–34. Cambridge Uni. Press, 1989.

[81] J. B. Long and A. D. Whitefield, editors. *Cognitive Ergonomics and Human Computer Interaction.* Cambridge Uni. Press, 1989.

[82] T. Maibaum. Temporal reasoning over deontic specifications. In *Deontic Logic in Computer Science — Normative system specification*, pages 141–202. Wiley, 1993.

[83] M. Mezzanotte and F. Paternó. Verification of properties of human-computer dialogues with an infinite number of states. In *Formal Aspects of the Human Computer Interface.* BCS-FACS, Springer Verlag, 1996.

[84] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture notes in computer science.* Springer Verlag, 1980.

[85] A. Newell. The knowledge level. *AI Magazine*, pages 1–20, 1981.

[86] A. Newell. *Unified theories of cognition.* Harvard University Press, 1990.

[87] A. Newell and S. K. Card. The prospects for psychological science in human-computer interaction. *Human-Computer Interaction*, 1:209–242, 1985.

[88] A. Newell and S. K. Card. Straightening out softening up: Responce to Carroll and Campbell. *Human-Computer Interaction*, 2:251–267, 1986.

[89] D. A. Norman. Cognitive Engineering. In *User Centered System Design – New Perspectives on Human-Computer Interaction*, pages 31–61. Lawrence Erlbaum Associates, 1986.

[90] P. Palanque and R. Bastide. Formal specification and verification of CSCW using the interactive cooperative object formalism. In *People and Computers X*, pages 213–231, 1995.

[91] P. A. Palanque and R. Bastide. Petri net based design of user-driven inetrfaces using the Interactive Cooperative Objects formalism. In F. Paternó, editor, *Proceedings of the Eurographics Workshop on Design Specification and Verification of Interactive Systems '94*, pages 215–228. Eurographics, 1995.

[92] F. Paternò and M. Mezzanotte. Analysing MATIS by interactors and ACTL. Technical Report SM/WP36, Amodeus project. ESPRIT BRA 7040, 1994.

[93] F. Paternó and P. Palanque. Formal methods and human-computer interaction. Comparisons, benefits, open questions. *SIGCHI Bulletin*, 28(4):46–48, 1996. Summary of CHI'96 workshop.

[94] S. G. Payne and T. R. G. Green. Task-action grammars: a model of mental representation of task languages. *Human-Computer Interaction*, 2(2):93–133, 1986.

[95] J. L. Peterson. *Petri net theory and the modeling of systems.* Prentice Hall, 1981.

[96] G. E. Pfaff, editor. *User Interface Management Systems.* EurographicSeminars. Springer-Verlag, 1985.

[97] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium of Foundations of Computer Science*, pages 46–57, 1977.

[98] A. Pnueli. System specification and refinement in temporal logic. *Lecture Notes in Computer Science, Vol 652*, pages 1–38, 1992.

[99] P. Ravn, H. Rischel, and K. Mark Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, 1993.

[100] P. Reisner. Formal grammar and design of an interactive system. *IEEE Transactions on Software Engineering*, SE–5:229–240, 1981.

[101] P. Reisner. What is inconsistency? In D. Diaper, D. Gilmore, G. Cockton, and B. Shakel, editors, *Human-Computer Interaction — INTERACT'90*, pages 175–181. Elsevier Science Publishers, 1990.

[102] C. R. Roast and M. D. Harrison. User centred system modelling using the template model. In F. Paternó, editor, *Proceedings of the Eurographics Workshop on Design Specification and Verification of Interactive Systems '94*, pages 261–273. Eurographics, 1995.

[103] M. Ryan, J. Fiadeiro, and T. Maibaum. Sharing actions and attributes and modal action logic. In T. Ho and A. Meyer, editors, *Proceedings on the International Conference on Theoretical Aspects of Computer Science*, pages 569–593. Springer Verlag, 1991.

[104] F. Schiele and T. Green. HCI formalisms and cognitive psychology: the case of task action grammars. In *Formal Methods in Human-Computer Interaction*, Cambridge series on HCI, pages 9–62. Cambridge Uni. Press, 1990.

[105] E. Schlungbaum and T. Elwert. Dialogue graphs — a formal and visual specification technique for dialogue modelling. In *Formal Aspects of the Human Computer Interface.* BCS-FACS, Springer Verlag, 1996.

[106] J. A. Simpson and E. S. C. Weiner, editors. *The Oxford English Dictionary.* Clarendon Press, Oxford, second edition, 1989.

[107] J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, 1989.

[108] G. Storrs. A conceptual model of human-computer interaction. *Behaviour and information technology*, 8(5):323–334, 1989.

[109] G. Storrs. A Conceptualisation of Human-Computer Interaction. In *IEE Colloquim on 'Theory in Human-Computer Interaction (HCI)' (Digest No.192).* IEE, 1991.

[110] G. Storrs. A conceptualization of multiparty interaction. *Interacting with Computers*, 6(2):173–189, 1994.

[111] G. Storrs. The notion of task in human-computer interaction. In *People and Computers X*, pages 357–365, 1995.

[112] B. Sufrin and J. He. Specification, analysis and refinement of interactive processes. In *Formal Methods in Human-Computer Interaction*, Cambridge series on HCI, pages 153–199. Cambridge Uni. Press, 1990.

[113] K. Systä. Specifying user interfaces in DisCo. *SIGCHI Bulletin*, 26(2):53–58, 1994.

[114] H. Thimbleby. On formal methods in HCI. In *IEE Colloquium on 'Formal methods in HCI: III' (Digest No. 151).* IEE, 1989.

[115] H. Thimbleby. You're right about the cure: don't do that. *Interacting with computers*, 2(1):8–25, 1990.

[116] J. C. Torres and B. Clares. Using an abstract model for the formal specification of interactive graphic systems. In F. Paternó, editor, *Proceedings of the Eurographics Workshop on Design Specification and Verification of Interactive Systems '94*, pages 275–292. Eurographics, 1995.

[117] R. M. Young, P. J. Barnard, A. E. Blandford, and M. D. Harrison. The unselected window scenario. Technical Report CP52, Amodeus ESPRIT BRA 7040, 1994.

[118] R. M. Young, T. R. G. Green, and T. Simon. Programmable user models for predictive evaluation of interface design. In K. Bice and C. H. Lewis, editors, *Proceedings of CHI '89: Human Factors in Computing Systems*, pages 15–19. Association of computing machinery, 1989.

Appendix A

A glossary of terminology

Action A state relationship describing the relationship between two states. An action is the unit of functionality in TLA.

Activity A description of how the state develops through time. Activities are assumed not to suffer from Zeno's paradox.

Assumptions Descriptions of parts of systems that already exist and cannot be built.

Behaviour The set of all activities that are legal for a system to perform.

Computation The unit of functionality in an RSSL specification. It consists of an internal and external state space and its behaviour is formally defined by a collection of state relationships which describe...

- when it is enabled,
- a read phase which copies values from the public space to the private and may update the value of the public space,
- a process phase where values in the private space are updated, and
- a write phase where the value of the public space is updated according to the value of the private space.

Closed system A system that is entirely self contained — it is uneffected by and has no effect on any entities external to the system.

Decoration A symbol super-scripted to a variable name in a state relationship.

Dense set A set of items that can be ordered and for any pair of distinct items from that set there is always another item from that set that lies between then according to the ordering.

Deontic logic A logic augmented with operators to describe whether actions may or must occur.

Device The kernel of an interactive system. Devices tend to be pre-programmed with instructions on how to respond to their environment.

Enabling condition A property describing when it is possible for a computation to occur.

Environment Sub-systems in a reactive system that can pass requests to the kernel and can expect an obligated response.

Fairness A condition that prevents computations from being locked out from occurring. The definition of fairness used in this work is that of strong fairness — if a computation is enabled infinitely often then it occurs infinitely often.

Filter A mathematical formula for combining interface and user effects into a single probability function.

Interactive system A reactive system where the environment is populated by users and the kernel by devices.

Interface effect The biasing to a use that is due purely to the user interface.

Interleaving function A function that takes the ordinal position of a computation in a computation sequence to the ordinal position of its read and write phase.

ISSL specification An Interactive System Specification Language specification. An ISSL specification is an RSSL specification such that each reaction has a probability function associated with it that describes the probability of a reaction occurring in given contexts.

Kernel Sub-systems in a reactive system that are obligated to respond to requests from the environment.

Liveness Part of a statement of requirements that expresses what the proposed system should eventually do.

Open system A system that can effect or be effected by entities external to the system.

Optimal behaviour Behaviour that gets a given task done as efficiently as possible.

Outcome How the public space is updated by the write phase of a computation.

Property A state relationship which expresses what is true about is single state.

Probability function A function that takes what has happened so far in a system and returns the probability of the user invoking a certain reaction next.

Raw activity An activity that may suffer from Zeno's paradox.

Raw ISSL specification An ISSL specification where reactions are mapped to pairs of probability functions — one representing the user effect and one the interface effect.

Reaction A pair of computations where the first computation is an optional environment computation which enables an obligatory kernel computation, thereby causing it to occur.

Reaction style RSSL specification A specification consisting of an initial property and a collection of optional reactions.

Reactive system A system that can be divided into sub-systems which can be put in two exclusive categories, the environment and the kernel.

Refinement The process of adding detail to an abstract description. This addition of detail constrains the behaviour of a described system whilst still ensuring that the system does everything that it must.

Requirements A description of the behaviour of a proposed system that makes as little reference to the mechanics of the system as possible. Requirements should be more about describing the problem the system is intended to overcome rather than the system itself.

RSSL specification A Reactive System Specification Language specification. An RSSL consists of an initial property and a collection of obligatory computations and a collection of optional computations.

Safety Part of a statement of requirements that expresses what proposed system should never do.

Side effect How the public space is updated by the read phase of a computation.

Specification A (probably) abstract description of a system. A specification can also be a description of sub-systems that need to be built, as opposed to assumptions which describe the existing sub-systems.

Specification non-determinism Non-determinism that is introduced into a specification due to abstraction, sometimes known as 'don't care' non-determinism. It is reduced by refinement.

State A snap-shot of the condition of a system at a particular instance. The state space for a system is a collection of variable names. The state of a system is an assignment of values to the variables in its state space.

State discontinuity A point at which the state changes.

State relationship An expression of what is true about a certain collection of states. In this work state relationships are expressed in a predicate logic notation with decorations added to the variables to denote which state their value is extracted from.

System A collection of entities that work together to produce some behaviour.

Temporal logic A logic that is augmented with operators to describe how time develops.

Truth valued function A mathematical entity which can be evaluated against a list of values to give a Boolean value.

Update A pair of times during which a most one state discontinuity occurs.

Update function A function that takes the ordinal position of a read or write phase to an update.

Usage distribution A graph that maps how 'good' an interaction is against how likely it is for an interaction that good to occur.

Use A probability distribution over what a system does. In this work we model use as a probability distribution over sequences of reactions. Alternatively we could model use as a probability distribution over a behaviour space.

User effect The biasing to a use that is caused purely by the user's intentions and motivations.

User interface An entity in an interactive system that alters the use (but not the behaviour) of a system.

User non-determinism Non-determinism in a discrete model of system behaviour caused by the specifier's inability to precisely determine what the user will do.

Zeno's paradox A philosophical difficulty encountered with real time models of behaviour. Time advances, but by infintessimally small increments and hence there is a time finitely in the future that is never reached.

Appendix B

Mathematical notation

B.1 Logical operators and constants

i	$\mathsf{true}, \mathsf{false}$	Logical constants
ii	$\neg P$	Negation
iii	$P \wedge Q$	Conjunction
iv	$P \vee Q$	Disjunction
v	$P \dot{\vee} Q$	Exclusive disjunction
vi	$P \Rightarrow Q$	Implication
vii	$P \Leftrightarrow Q$	Biconditional (iff)
viii	$\exists x \bullet P$	Existential quantification
ix	$\forall x \bullet P$	Universal quantification

B.2 Sets

i	$\{1, 2, 3\}$	Explicit set
ii	$\{x{:}X \mid P(x)\}$	Implicit set — all the elements of type X that satisfy the predicate P. If the type of x is clear through context then it may be omitted.
iii	$x \in X$	Set membership — x is a member of set X.
iv	$\mathbb{B}$	Boolean values — $\{\mathsf{true}, \mathsf{false}\}$
v	$\mathbb{N}$	Natural numbers — $\{0, 1, 2, 3, \ldots\}$

vi	$\mathbb{N}_1$	Non-zero natural numbers — $\{1, 2, 3, \ldots\}$
vii	$\mathbb{R}$	Real numbers — $[-\infty, \infty]$
viii	$m \mathinner{..} n$	Enumeration — $\{i{:}\mathbb{N} \mid m \leq i \leq n\}$
ix	$\mathcal{P}(X)$	All subsets of X
x	$\mathcal{F}(X)$	All finite subsets of X
xi	$X \times Y$	Cross product of X and Y — $\{(x, y) \mid x \in X \land y \in Y\}$
xii	$X \subseteq Y$	X is a subset of Y
xiii	$X \subset Y$	X is a proper subset of Y
xiv	$X \cup Y$	Set union — $\{x \mid x \in X \lor x \in Y\}$
xv	$X \cap Y$	Set intersection — $\{x \mid x \in X \land x \in Y\}$
xvi	$X \setminus Y$	Set minus — $\{x \mid x \in X \land x \notin Y\}$

B.3 Bags (or multisets)

i	$\{\!\mid x, x, y, z \mid\!\}$	Explicit bag
ii	$x \in X$	Bag membership
iii	$\mathcal{B}(X)$	All subbags of X (where X is a set)
iv	$X \subseteq Y$	X is a sub-bag of Y
v	$X \uplus Y$	Bag union
vi	$X \setminus Y$	Bag minus

B.4 Functions

i	$X \xrightarrow{p} Y$	Partial function from X to Y — if $f{:}X \xrightarrow{p} Y$ then...
		$f{:}\{(x \mapsto y) \mid x{:}X \land y{:}Y \land \exists z \bullet (x \mapsto z) \in f \Rightarrow y = z\}$
ii	$f(x)$	Function application — $f(x) = y$ such that $(x \mapsto y) \in f$
iii	$\mathbf{dom}\ f$	Function domain — $\{x \mid \exists y \bullet (x \mapsto y) \in f\}$
iv	$\mathbf{ran}\ f$	Function range — $\{y \mid \exists x \bullet (x \mapsto y) \in f\}$
v	$X \rightarrow Y$	Total function from X to Y. If $f{:}X \rightarrow Y$ then $\mathbf{dom}\ f = X$
vi	$f_1 \oplus f_2$	Function overwrite. $(f_1 \oplus f_2)(x) = f_2(x)$ is $x \in \mathbf{dom}\ f_2$, $f_1(x)$ otherwise.

B.5 Sequences

i	$\langle a, b, c \rangle$	Explicit sequence
ii	X^*	All finite sequences of elements of X. A finite sequence $s : X^*$ is a partial function from $\mathbb{N}_1$ to X such that *contig*(**dom** s). (The predicate *contig* is defined in the next section.)
iii	X^ω	All infinite sequences of elements of X An infinite sequence $x : X^\omega$ is a total function from $\mathbb{N}_1$ to X
iv	X^Ω	All finite or infinite sequences of X — $X^* \cup X^\omega$
v	$\lvert S \rvert$	Length of finite sequence S
vi	$S(n)$	nth item of sequence S $(1 \leq n \leq \lvert S \rvert)$
vii	$S_1 \frown S_2$	Sequence concatenation (S_1 must be finite)

B.6 Some miscellaneous functions and predicates

The functions *max* and *min* respectively extract the largest and smallest items from a non-empty set of natural numbers.

$$\begin{array}{l} max : \mathcal{P}(\mathbb{N}) \to \mathbb{N} \\ max(ns) \triangleq n \\ \textbf{where}\ \ n \in ns \wedge \exists x : \mathbb{N} \bullet x \geq n \wedge x \in ns \Rightarrow x = n \end{array} \tag{B.1}$$

$$\begin{array}{l} min : \mathcal{P}(\mathbb{N}) \to \mathbb{N} \\ min(ns) \triangleq n \\ \textbf{where}\ \ n \in ns \wedge \exists x : \mathbb{N} \bullet x \leq n \wedge x \in ns \Rightarrow x = n \end{array} \tag{B.2}$$

In words; 'the function *max* returns a number n from the set ns such that if there is number x also in ns that is greater of equal to n then $x = n$. Similarly *min* returns n such that if x is less of equal to n then $x = n$.'

The following list processing functions are defined; *hd*, *tl*, *lt* and *ft* which return the head of a sequence, the tail of a sequence, the last item in a sequence and the front of a list respectively.

$$\begin{array}{l} hd : X^\Omega \to X \\ hd(\langle x_1, x_2, x_3, \ldots \rangle) \triangleq x_1 \end{array} \tag{B.3}$$

$$tl : X^{\Omega} \rightarrow X^{\Omega}$$
$$tl(\langle x_1, x_2, x_3, \ldots \rangle) \hat{=} \langle x_2, x_3, \ldots \rangle \tag{B.4}$$

$$lt : X^{*} \rightarrow X$$
$$lt(\langle \ldots, x_{n-2}, x_{n-1}, x_n \rangle) \hat{=} x_n \tag{B.5}$$

$$ft : X^{*} \rightarrow X^{*}$$
$$ft(\langle \ldots, x_{n-2}, x_{n-1}, x_n \rangle) \hat{=} \langle \ldots, x_{n-2}, x_{n-1} \rangle \tag{B.6}$$

pref and *suff* take a sequence and return sets of sequences that are prefixes and suffixes of the given sequence.

$$pref : X^{\Omega} \rightarrow \mathcal{P}(X^{*})$$
$$pref(seq) \hat{=} \{start : X^{*} \mid \exists end : X^{\Omega} \bullet seq = start^{\frown} end\} \tag{B.7}$$

$$suff : X^{*} \rightarrow \mathcal{P}(X^{*})$$
$$suff(seq) \hat{=} \{end : X^{*} \mid \exists start : X^{*} \bullet seq = start^{\frown} end\} \tag{B.8}$$

In words; 'the set of all prefixes of *seq* is the set of all sequences *start* such that there is a sequence *end* which concatenated to *start* gives *seq*. Similarly the set of all suffixes of *seq* is all the sequences *end* such that there is a sequence *start*,'

The predicate *contig* holds true if the given set contains all the natural numbers between 1 and the maximum value in the set, or contains all the non-zero natural numbers if the set is infinite.

$$contig : \mathcal{P}(\mathbb{N}_1) \rightarrow \mathbb{B}$$
$$contig(ns) \hat{=} (\exists n : \mathbb{N}_1 \bullet n = max(ns) \wedge ns = \{1 \ldots n\}) \vee (ns = \mathbb{N}_1) \tag{B.9}$$

In words; 'if there is a maximum value in ns then all numbers from 1 to n are in ns. If ns is infinite then all the non-zero natural numbers are in ns.'