UNIVERSITY
of
GLASGOW

Department of Computing Science

Declarative Support for Prototyping Interactive Systems

Meurig Sage

A thesis submitted for a Doctor of Philosophy Degree in Computing Science at the University of Glasgow

March 2001

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

The development of complex, multi-user, interactive systems is a difficult process that requires both a rapid iterative approach, and the ability to reason carefully about system designs. This thesis argues that a combination of declarative prototyping and formal specification provides a suitable way of satisfying these requirements.

The focus of this thesis is on the development of software tools for prototyping interactive systems. In particular, it uses a declarative approach, based on the functional programming paradigm. This thesis makes two contributions. The most significant contribution is the presentation of FranTk, a new Graphical User Interface language, embedded in the functional language Haskell. It is suitable for prototyping complex, concurrent, multi-user systems. It allows systems to be built in a high level, structured manner. In particular, it provides good support for specifying real-time properties of such systems. The second contribution is a mechanism that allows a formal specification to be derived from a high level FranTk prototype. The approach allows this to be done automatically. This specification can then be checked, with tool support, to verify some safety properties about a system. To avoid the state space explosion problem that would be faced when verifying an entire system, we focus on partial verification. This concentrates on key areas of a design: in particular this means that we only derive a specification from parts of a prototype. To demonstrate the scalability of both the prototyping and verification approaches, this thesis uses a series of case studies including a multi-user design rationale editor and a prototype data-link Air Traffic Control system.

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisors Professor Chris Johnson and Professor Simon Peyton Jones for their guidance, enthusiasm and support during this research. I would also like to thank Conal Elliott for introducing me to Fran and Functional Reactive Programming, and for answering my questions so enthusiastically. Finally, I would like to thank Tab Lamoureux and the UK's National Air Traffic Services for cooperating on the Air Traffic Control case study.

This research would not have been possible without financial support from EPSRC. I am also grateful to Simon Peyton Jones and Microsoft Research for employing me, during the summer of 1999, as an intern, to work on FranTk. I would also like to thank Phil Gray for giving me time off to complete the writing up of this thesis.

Finally I'd like to thank family and friends for giving me the support and encouragement necessary to finish this thesis.

Declaration

I hereby declare that this thesis has been composed by myself, that the work herein is my own except where otherwise stated, and that the work presented has not been presented for any other university degree before.

Sections 2.4, 6.3 and 11.5 contain revised versions of material published in [173]. Chapter 4 contains material published in [174]. Finally, Appendix C contains material previously published in [169] and [170].

Meurig Sage

Part I. Introduction

The first part of this thesis discusses why high level prototyping and formal verification are both necessary (though not sufficient) for the design of complex, interactive systems. It argues that a mixture of visual and declarative programming provide a powerful mechanism for interactive system development. It also argues that formal verification is best applied later on in the design process once an initial design has been developed.

Chapter 1 - Introduction

1.1. Interactive System Design

The development of complex, multi-user interactive systems requires high levels of time and expertise. Surveys of programming project [138] have shown that it is not uncommon to spend 50% of the resources of a project developing the user interface. This is because it is generally impossible to produce an interactive system that does everything required, first time round. Producing usable systems requires an iterative approach with a reliance on user testing. There are principles, such as usability heuristics [142] which can be used to encourage good design. However, even when developing relatively simple systems, they cannot guarantee good products.

The development and evaluation of multi-user "Computer Supported Co-operative Working" (CSCW) systems is a particularly difficult problem. Such systems generally allow group awareness and support co-ordination and communication between different users. These users must be able to understand the common context that they are working within. Grudin[79] has argued that many early attempts at developing CSCW systems failed because they did not map end user requirements to appropriate co-ordination mechanisms. This means that usability heuristics will be even more difficult to use in a CSCW project. Only through thorough evaluation can we hope to develop systems that truly support their users. Good examples of such problems arise in the development of Air Traffic Control (ATC) systems. Social studies of such systems [84] have shown that co-ordination between Air Traffic Control Officers is subtle, complex, and often outwith the bounds of regulated procedures. The usability of these ATC systems is sometimes in direct contrast to standard usability heuristics. Storrs and Windsor [190], for instance, describe an ATC information display that seemed "conceptually difficult and noisy", but which controllers found more useful and usable than other "simpler" designs. A development approach based around iterative, user-centred design is therefore important when developing such systems, because designers' intuitions cannot be relied upon.

Much of the research to aid user interface construction, has therefore been directed at providing rapid prototyping systems[71]. This thesis builds on this research. It is concerned with software techniques to support rapid prototyping. In particular, it is concerned with providing support for the development of novel interfaces for complex, dynamic, multi-user systems. The first and most significant contribution of this thesis, is the presentation of FranTk, a new functional Graphical User Interface Language, that is suitable for prototyping complex, concurrent, multi-user systems. It allows systems to be built in a high level, structured manner. In particular, it provides good support for specifying real-time properties of such systems.

An additional requirement with such interfaces is the need to reason about concurrent behavior within the interactive system. The complexity of large systems, such as those used in Air Traffic Control, makes it difficult to test them fully. A secondary contribution of this thesis is a mechanism for automatically deriving a formal specification from the high level prototype. This specification can then be checked, with tool support, to verify some safety properties about a system.

This thesis demonstrates the validity of these techniques through the use of three large case studies. These case studies represent three different levels of complexity, culminating in the development of a multi-user, real-time, data-link Air Traffic Control simulator.

1.2. The Software Engineering Lifecycle

The domain of Software Engineering provides a number of approaches to developing large software systems, such as Air Traffic Control systems. There are a number of different software lifecycles which consider how we should develop these systems. The first of these was the "waterfall model" [187], which assumed a steady progression from requirements, to design, to development, to testing and evaluation, to maintenance. There have been numerous more recent developments of this model, which attempt to encourage early evaluation and a more iterative approach to design. These tend to encourage the use of prototyping in the requirements and design phase. Organisations such as the UK's National

Air Traffic Services – which is responsible for the design of new Air Traffic Control Systems – carry out detailed iterations through requirements, design, prototyping and evaluation phases before handing off a final design to a software development company. This thesis concentrates on providing tools for use in the prototyping phase. It does not concern itself with the final implementation and testing of such large systems.

1.3. Rapid Prototyping

1.3.1. The Need For Rapid Prototyping Programming Tools

Rapid prototyping tools must allow interactive systems to be developed at an appropriate level of abstraction. Developers should not be bothered by low-level programming details when attempting to develop a prototype. This would only slow down an already difficult process. However, tools must leave enough control in the hands of the developer for them to develop the interface that they desire.

It has been argued that rapid prototyping tools can be subsumed by a model based approach to development. These attempt to actually derive the interface design from a selection of models. One of the earliest, and most well-known of these model based approaches was Adept [103]. Adept was a proof-of-concept prototype that would automatically generate an interface from a detailed task model. In order to carry out this transformation it also made use of a very simple user model which described user knowledge in terms of preferences for different styles of interface (e.g. menus versus forms).

There are a number of fundamental problems, however, with trying to automatically generate interfaces from task based models. An approach such as task modelling is only capable of accurately considering user activity at a very high level of abstraction. Adept removed control over the interface appearance from the developer. Draper [40] argues that trying to specify plans for user behaviour down to a low level can be dangerous:

> "It would seem then that we cannot expect fixed and predictable behaviour from human users even at quite "low" levels. Therefore whenever the device allows any variation in method, task analyses are not likely to work at low levels."

Instead of trying to restrictively define what a user should do for each task, he argues that we should create flexible interfaces that allow users to work in a variety of ways. A task model should therefore be used in association with other contextual information to design a system. Therefore automatically generating an interface from a task model may not be very effective.

Fields and Merriam [58] also argue that using a purely task-based approach can make it difficult to consider important issues. They argue that an "information resources" based approach may also be required, especially when considering complex multi-user domains such as Air Traffic Control. Task based approaches tend to lead to action oriented designs, that consider only what a user must do. However, many activities such as Air Traffic Control involve monitoring information. In these cases, we must also consider what information users need to understand the behaviour of a given system. One of the major focuses of design here is therefore to consider how particular information resources will be presented to the user. Model based designs will not be very effective here if they try to abstract away from these presentation issues.

Because of these restrictions, model based approaches are too restrictive to apply to dynamic, complex interfaces. They remove too much control over interface design from the developer. While modelling can be used as part of the design process, a prototype interactive system still needs to be developed separately, using some other technique.

1.3.2. Visual Programming

Visual approaches, that allow interfaces to be built by direct manipulation, represent a popular declarative approach. Developers can define what an interface should look like, rather than saying how it should be produced. Many visual based programming languages exist for developing user interfaces. Visual toolkits are commonly provided to allow a developer to graphically build an interface. These work well when applied to static interactive system, such as simple form based interfaces. It is usually

easier to draw such an interface than to program it. When developing standard application behavior, such as "Print Dialogues", development tools can provide application frameworks which fill in the appropriate code. However, when developing novel, dynamic interactive systems it is still necessary to write much of the interface and application using textual code.

Visual approaches have also been used to construct the architecture of an interactive system. The Clock language uses such an approach[71]. Visual architectures can make it easier to explain the structure of a system to a non-programmer. They can also make it easier to understand the structure of a large system. For instance, class diagrams are commonly used in object oriented programming to provide an overview of a system, as they summarise interaction between system components. Viewing such a diagram is clearly easier than trawling through pages of code.

1.3.3. Declarative Programming

This thesis is concerned with declarative programming approaches, and in particular with the functional programming paradigm. Declarative programming approaches allow a developer to specify what a program should do, not how it should do it. When developing user interfaces, a traditional imperative approach forces a programmer to state how both the application state and the interface change, on every input, as a series of actions. In contrast, in a declarative approach the programmer should provide one definition of the appearance of an interface component that will describe its appearance for the duration of the program. Functional programming emphasises composition. *Values* representing programs are constructed by combining smaller units. A functional language for developing user interfaces should therefore consider user interface components as values and allow them to be easily composed.

Over the last few years there has been a great deal of interest in the development of functional GUI libraries. These have used a number of different mechanisms to allow programmers to structure their code. Some such as TkGofer [204] have used traditional callback based approaches. These make it difficult to structure complex interactive systems as the structure of the application is turned inside out [137]. The application cannot call the GUI library, instead the application must be called by the library. A more popular solution has been the use of imperative concurrency [62]. This style allows an application to be structured as a number of threads that execute concurrently and consume user input. However, these approaches require a programmer to handle the intricacies of full concurrent programming, dealing with mutual exclusion and race conditions between processes. These approaches all force a programmer to use a very imperative style of programming; an unfortunate requirement in a declarative language.

In contrast, the User Interface Management Community has been investigating the use of "constraint" based approaches for programming interactive systems [136]. Here an application is defined as having a behaviour, responding to user input and updating its state. The appearance of the interface is defined as a function of the application's state. This allows a programmer to say what an interface should look like, rather than saying how it should be implemented. This style is particularly powerful when defining multiple views of the same state. One of the most well-known of these systems, Garnet [136], was implemented in Lisp. However, it still relied on side-effects to implement changes in the interface.

The languages developed in these two communities both use an imperative style of programming. The difference has been described as being "Declarative in the Small versus Declarative in The Large" [71]. The languages developed in the functional programming community have been declarative in the small, allowing individual aspects of a system to be written in a purely functional style. However, they have been imperative in the large, forcing programmers to structure their programs as a set of imperative actions. In contrast, the user interface management community has concentrated on providing systems that are declarative in the large, allowing user interfaces to be structured as a set of constraints or functions. However, being based largely in imperative languages they have not provided the advantages of higher order functions, and referential transparency when building individual components.

There have been a few notable attempts to overcome these difficulties and combine the advantages of both. These include Clock [71] and Pidgets [178]. One other important language seems to lend itself to this style of programming. Fran [44] (Functional Reactive Animation) is a language developed for constructing interactive animations. It uses a high-level modelling approach which allows programmers to describe what an animation should look like, not how it should be implemented. It introduced a style

of programming known as Functional Reactive Programming (FRP) which has two key notions: *behaviors* and *events*. *Behaviors* are time-varying, reactive values, while *events* are streams of values that occur at a specific times.

This thesis draws from ideas in Fran and Clock to provide FranTk, a high level language for interactive systems. The emphasis of this thesis is very practical. It concerns the development of a fully fledged user interface library. Importantly, it is also concerned with applying such a library to significant case studies. Though many toolkits have been developed, they are most often only evaluated in the context of small examples. The case studies used in this thesis were chosen to be representative of larger real world programs. In particular, these case studies have been chosen to demonstrate highly interactive, dynamic, multi-user, real-time systems.

1.4. Formal Modelling

Formal specifications can be used to help develop interactive systems. They can be used to design systems, and allow developers to prove the functional correctness of their systems. For instance, we could prove that a design meets specific, formally defined, requirements. This is important because incremental development, based simply on prototyping and testing, cannot guarantee certain critical system properties. When designing a system, there may be millions of possible system states. No amount of testing can significantly test such large state spaces. The problem becomes particularly significant when we wish to prove negative properties about a system. For instance, when designing a rail track control system, we may wish to prove that "A route will never be set if conflicting routes are set" [82]. This sort of critical requirement is impossible to prove simply by user testing, given any significant system. Though many of these critical requirements are what can be termed functional (unrelated to the interface), many others will be related to user interactions. For instance, when developing an air traffic control system there will be certain interaction requirements that will be critical, such as "A control order can be sent to only one plane" [149].

Formal analysis can be used to verify completeness criteria about user interaction, to search for paths to hazardous states that might be reached within an interface, and to verify consistency questions about interaction when in different modes of a system [119].

Formal specifications have also been used to reason about usability properties of a system [85]. Various interaction concepts have been suggested. These include *predictability*, whether a system behaves as expected; *visibility*, whether the necessary information is displayed to allow users to act successfully; *continual feedback*, whether a systems provides the necessary feedback to allow users to understand their actions; and *reachability*, whether a user can get to all states in a system, or whether they could get stuck in an interaction deadlock. These principles, though important, are very general. This makes them difficult to check. The use of formal methods to prove the usability of an interface is a difficult, and troubled issue. The notion of usability itself is still a difficult topic. The concept of usability, especially when dealing with a multi-user system, can only be understood in the context of the application and user's work

This thesis presents a method that allows a formal LOTOS specification to be derived from a structured FranTk prototype. The model can be analysed to verify important safety properties about the system. This approach has been evaluated using the case studies.

1.5. Thesis Contributions

The major contributions of this thesis fall into two categories: those related to FranTk, and those related to formal verification.

1.5.1. Contributions to Declarative GUI languages

The most significant contribution of this thesis is the presentation of FranTk, a new functional GUI language, embedded in Haskell[158]. *FranTk was designed to be used by programmers who are*

familiar with functional programming[1]. It improves on previous functional GUI languages by supporting a style of programming that is closer to the goal of being both "Declarative in the Large and in the Small". By doing this, it attempts to provide a more compositional style of programming than has previously been possible in such languages.

This thesis presents a more efficient implementation of the core Functional Reactive Programming combinators. This is significant because there are a growing number of other application areas to which the FRP approach has been applied. These currently include robotics[155], multimedia[200] and animation[44]. Current work at Yale University is investigating its application to Vision systems. The search for more efficient FRP implementations is therefore important to all of these application areas.

It is important to note here that this thesis *does not* attempt to evaluate the usability of FranTk itself. This would be a very significant task in its own right. Performing such evaluations has proved difficult [92]. The usability of any language is heavily dependent on the skills of the programmer, and their experience with similar languages. Performing such an evaluation would be fraught with difficulty and is therefore well beyond the scope of this thesis.

Instead this thesis evaluates FranTk in terms of a set of significant case studies. This is important because only through such case studies can we determine how well a language scales to real-world problems. These case studies therefore demonstrate only how well the designer of the language was able to use it. FranTk does, however, have a number of other users. Chapter 6 will briefly discuss some of their comments about using the system.

Through the design and implementation of FranTk, this thesis makes the following contributions.

1.5.1.1. FranTk Design

- FranTk lifts Fran's behaviors and events to widgets. This is the key to the declarative style of programming. The appearance of a widget can be defined in one function, *for all time,* in terms of FranTk combinators. An interface can therefore be defined as a function of some application state.

- FranTk provides good support for dynamic in addition to static interfaces. The construction of systems with dynamically changing number of components can be difficult in many GUI systems, and frequently requires a very imperative and sometimes cumbersome style of programming. The use of behavioral values and dynamic collections allows a single abstract model of an application to be produced. We can then have multiple views of this model, providing good application/interface separation.

- FranTk extends Fran with support for hierarchical interactive displays, allowing access to input from individual components rather from one monolithic window. This is vital to allow a truly compositional style of programming.

- FranTk separates visual composition from semantic wiring. These two concepts are fundamental to GUI programming. The first involves geometric composition. For instance, placing one widget above another. The second involves connecting user input from a widget to the application code. This separation is made possible by the introduction of *listeners*, consumers that respond to user input. FranTk provides an algebra to compose these listeners in a functional style. This separation allows a more compositional style of programming.

- This thesis presents two visual tools, an architecture tool and a static widget construction tool, which demonstrate how the advantages of visual programming could be incorporated into FranTk.

1.5.1.2. FranTk Implementation

- This thesis presents three novel, clever implementations of the core Functional Reactive Programming combinators. Each implementation is significantly more efficient than the simple

[1] Readers without the necessary background may wish to refer to [94] or [199] or [15], all of which provide a good introduction.

streams implementation provided in the original versions of Fran[46]. Each implementation relies for efficiency on two key features.

- *Data Driven Behaviors and Events.* The streams implementation of events and behaviors requires that behaviors and events are sampled every time interval. This would be prohibitively expensive in a large user interface, as every aspect of the interface would need to be redisplayed every time any input was received. Instead FranTk, uses a data driven model. Events and behaviors have invalidation actions associated with them. After any user input only those components that rely on behaviors or events that have been invalidated need to be redrawn.

- *Weak Listeners and Finalisers.* A simplistic implementation of the FRP combinators can easily result in serious time and space leaks. Behaviors are updated by *listeners* as a result of user input. This is useful only so long as the behavior is actually being used. However, often behaviors will only be used for a fraction of the lifetime of a program. For instance, if a component were later removed from the screen and the behavior it relied upon was no longer used it would be useful to remove the listeners which update it. For this purpose, FranTk uses weak references and finalisers. These allow listeners to be deleted when they are no longer needed, avoiding the potential time and space leaks.

- FranTk makes one further implementation contribution. It provides an efficient implementation of *incremental dynamic collections.* Fran provides behavioral values. These could be used to represent behavior collections of objects. For instance, we could display a dynamic list of objects. However if we were to render such a behavior collection, each time an element were to be added *the entire collection would have to be redrawn.* This would be prohibitively expensive if we needed to continually recreate complex compound collections. FranTk's incremental behavioral collections overcome this problem. They can be both viewed as a behavior and efficiently and incrementally rendered.

1.5.2. Contributions to Formal Verification

The secondary set of contributions relate to the generation of formal models of interactive systems. This thesis does not attempt to use formal models to understand the usability of a system. This has been attempted by a number of others including Campos[23], Rushby[168], Leveson[119]. Instead it concentrates on the use of formal methods to verify critical, application specific requirements. In particular, this thesis takes one restricted view of formal methods. It assumes that they should be used to find problems in a system, not to prove it correct. This thesis concentrates on providing an approach which *supports formal verification, of complex, domain specific properties by formal methods experts.*

In the field of formal modelling of interactive systems, this thesis makes one contribution:

- It presents a transformation mechanism that supports the creation of a formal, LOTOS, specification, which given certain parameters can be derived automatically from a structured FranTk prototype. This allows the generation of a formal model at relatively low cost. The model can be analyzed to verify important safety properties about the system design. To make the verification practical we focus on partial verification, focusing on critical areas of the design. This avoids the state-space explosion problems faced when trying to perform exhaustive proofs about a whole system.

This approach has been evaluated using the Air Traffic Control case study. The need for significant case studies was very important. Only through the use of a significant, safety critical case study, such as the Air Traffic Control system, can the utility of such an approach be demonstrated.

1.6. The Structure of this Thesis

This thesis is structured into 5 distinct parts. Part I contains this introduction. It also contains Chapter 2, which introduces the three case studies used in this thesis.

Part II of this thesis presents the design of FranTk, a functional graphical user interface library. Chapter 3 discusses previous approaches to the declarative development of interactive systems. This chapter also presents a set of requirements for the development of FranTk. Chapter 4 discusses the design of FranTk, demonstrating its important features through the use of a range of small examples. Chapter 5 presents two visual tools that were developed to help with the construction of interactive systems in FranTk. Chapter 6 evaluates FranTk in the context of the three case studies, showing how it supports the requirements in Chapter 3.

Part III of this thesis discusses the implementation of FranTk. Chapter 7 discusses the important issues that arose when implementing the core Functional Reactive Programming combinators. It also discusses the implementation of dynamic collections. Chapter 8 discusses the FranTk GUI library highlighting how it achieves a toolkit independent implementation.

Part IV discusses formal verification work. Chapter 9 discusses previous approaches to formal modelling of interactive systems. Chapter 10 presents the transformation mechanism that can be used to convert elements of a FranTk program into LOTOS. Chapter 11 discusses formal verification techniques in LOTOS, and evaluates the use of this approach with respect to the case studies.

Part IV contains Chapter 11, which presents conclusions and areas of further work.

Finally, this thesis contains three appendices. Appendix A introduces the basic concepts and functions provided by Functional Reactive Programming. These are the combinators for Behaviors and Events. Appendix B contains a brief discussion about usability evaluation in the design of interactive systems. It discusses a small evaluation that was carried out when developing the QOC case study. Appendix C discusses an earlier attempt to link formal specifications with functional GUI languages.

While this thesis may be read simply from start to end, there are several other possible routes through it. Readers only interested in using FranTk should concentrate on Part II: in particular Chapter 4, which discusses the FranTk design. Readers without an understanding of Functional Reactive Programming may also which to refer to Appendix A to gain an understanding of the basic FRP combinators. Readers interested in FranTk may also wish to read Chapter 5 which discusses the visual tools. They should then read Chapters 2 and 6 which discuss the case studies and the use of FranTk within them. Such readers may also wish to read Chapter 3 to understand how FranTk fits into the broader family of Graphical User Interface languages. Readers interested in the new Functional Reactive Programming implementation need to read only Chapter 7. Readers only interested in the widget implementation, such as those wishing to port FranTk to a new toolkit, should read Chapter 8. Readers only interested in the formal verification work should read Part IV of this thesis and Chapter 2 to understand the case studies.

Chapter 2 - The Case Studies

2.1. Introduction

In order to develop and evaluate the approach discussed in this thesis, I used a series of case studies. Declarative programming languages and formal verification techniques are frequently only applied to very small-scale examples. One of the most significant factors about the work in this thesis is that I have applied the approaches developed to three case studies, each increasingly complex. Demonstrating the scalability of the approaches was of prime importance.

2.2. The Space Game

The first case study ([169],[170]) involved the development of a highly interactive, real-time user interface. It is a space ship game, as shown in Figure 1. The user inputs commands via the keyboard. A number of enemy ships fly in waves across the screen. These destroy the player's ship if they collide with it. The player must avoid hitting the hills at the bottom, and the enemy ships. The aim is for the player to destroy the enemy base when it is finally reached, while shooting as many enemy ships as possible. There are buttons to allow the user to pause the game, restart it, or quit from it. The current score will be displayed on the screen. Though the graphics are only simple, the game requires real-time animation. This example therefore provides a highly reactive system, with a number of different interacting components. The example does not, however, have any significant notion of application/interface separation. There is only one view of any of the data and so there is no real need to be able to provide an abstract model of the game's progress.

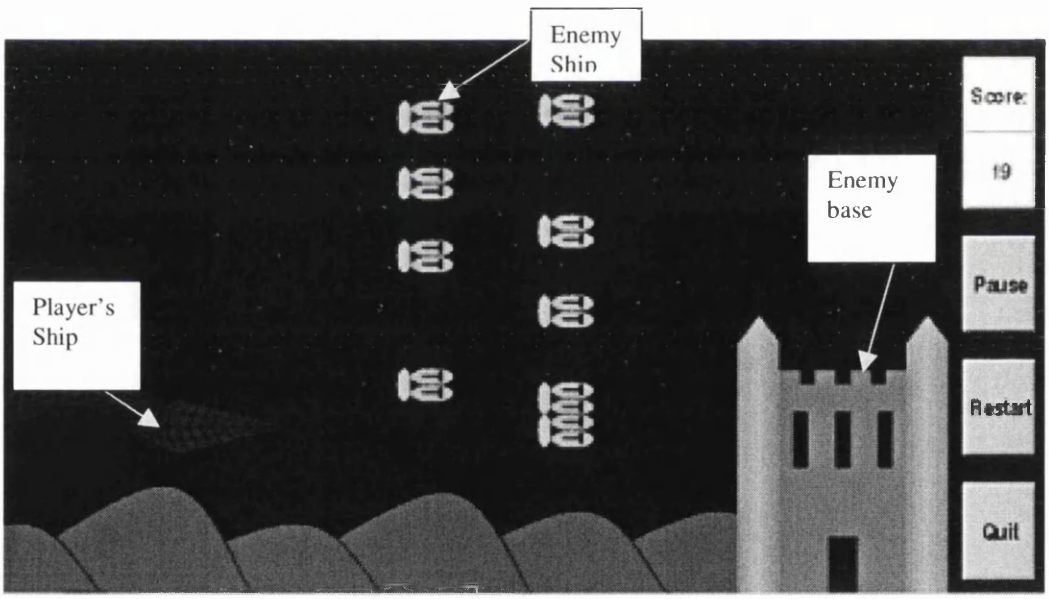

Figure 1 - The Interactive Game

2.3. The QOC Editor

2.3.1. Introduction

The second case study ([171]) involved the design of a multiple user, design rationale editor. This provided for a fairly complex case study, where significant design decisions were needed and where concurrency was required. This concurrency added an extra level of complexity that began to make formal specification and proof helpful. For instance, in ensuring that locking properties are satisfied for multiple users of a shared workspace

2.3.2. Design Rationale

Design rationale has received a lot of attention [133]. A number of semi-formal notations have been developed that attempt to document clearly why design decisions were made. The Questions, Options and Criteria (QOC) notation is one such notation developed at Rank Xerox [183]. It is a graphical notation that highlights key questions in a design, and links them with possible options and criteria that support those options. Several studies (e.g. [183]) have highlighted the need for tool support for this notation. A variety of tools have been developed, frequently based on hypertext systems. However, current tool support is frequently inadequate for designers' needs [185]. In particular, tools often provide little support for multi-user activities. Buckingham Shum argues that design rationale itself is still in its infancy. This makes it difficult to be sure exactly how designers will wish to use these tools. Iterative development is therefore required to explore different ways of satisfying the needs of designers.

2.3.3. Collaborative Software

This case study was appropriate because the development of collaborative software of this form is still in its infancy. It is frequently difficult to determine exactly how a group of users may wish to collaborate using a piece of software. It is very easy to produce software that does not properly consider how a group of users may share a design, and so seriously hinder the use of such a tool [13]. A design therefore needs to be well thought out, and will frequently go through several iterations before it can be useful. It can also be difficult to produce software for several users because of the concurrency involved. Complex locking mechanisms may be required that need serious thought [38]. A design therefore needs to be well structured.

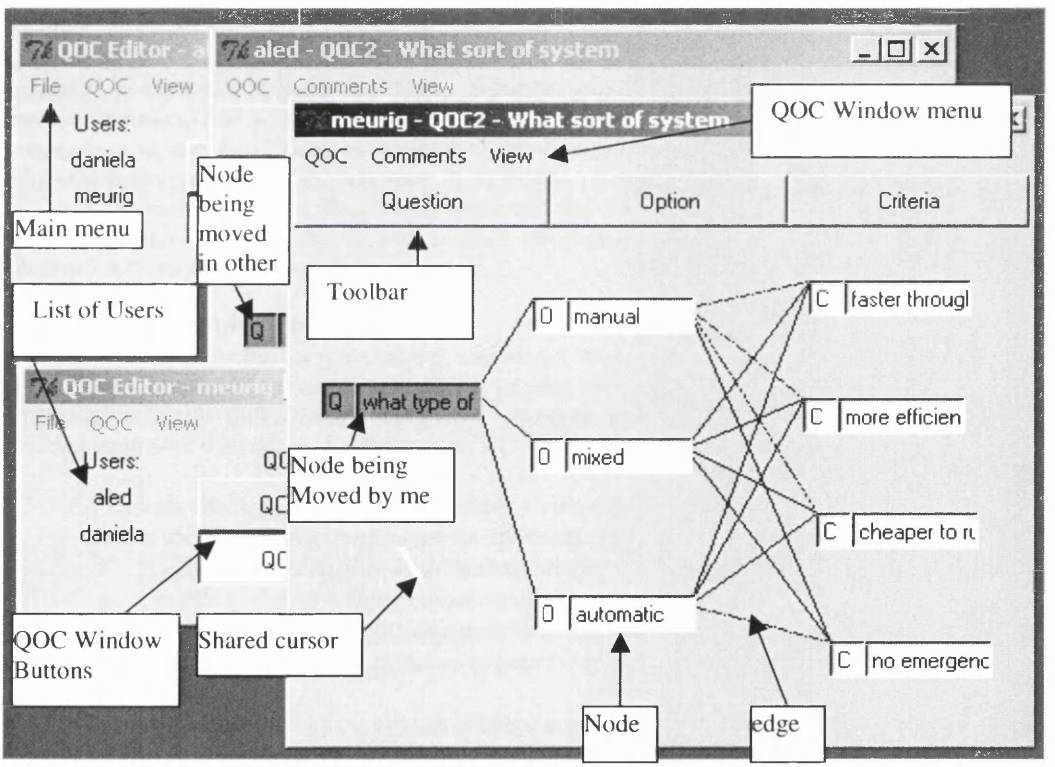

Figure 2 - The QOC Editor

2.3.4. The Prototype System

Figure 2 shows the interface for the prototype system we produced. It allows several users to build a QOC rationale. Each user has a separate view of the QOC collection. Each QOC is maintained in a window. Different users can have windows open in different areas of the screen. Within each window, however, users' views are strictly WYSIWIS (What-You-See-Is-What-I-See). Users can also note their

actions, and any extra textual information using a shared log. They can filter their own view, using the view menu. They can filter their view of nodes to show only Questions and Decisions; Questions, Decisions and Criteria; Questions and all Options; Questions, all Options and Criteria.

The level of sharing is important. In their current window, users can see changes made by anyone as they happen. In contrast, they will only see the results of changes in each of the other visible windows. They can also see where the other users' cursors appear in their own window. Users have a colour associated with them, used for their cursor. A different colour scheme is used to represent changing objects. Objects being edited by a user appear in green; objects being edited by another user appear in red. Locking is at node level, so that two users can both act in the same window, but cannot both act on the same node in the window simultaneously.

This example introduced some new requirements that were unnecessary in the "Space Fighter Game". In particular, it has a real need for separation between application and interface code. We require one explicit underlying model of the application, so that we can provide multiple views of the same data and so that we can save and load the contents of the QOC to and from files.

The prototype system was evaluated through a multi-user usability trial (discussed in Appendix B). This study was designed to formatively evaluate the case study. After each trial, incremental changes were made to the interface to fix problems experienced that were discovered. This allowed me to test the support for redesign that existed within the prototyping environment, and demonstrate whether the interface developed in such a declarative system would be efficient enough to really use.

2.4. The ATC System

The third and most significant case study ([172], [173]) involved the development of a prototype data-link Air Traffic Control System. This system was developed in association with a human factors specialist, at the UK's National Air Traffic Services, who provided the necessary domain knowledge and requirements. This section provides a detailed introduction to Air Traffic Control, discussing the processes used, and the difficulties involved in developing such systems. This discussion demonstrates why new prototyping tools are important. It then goes on to present the prototype system developed, summarising its functionality. This demonstrates that the ATC system formed a significant case study. Readers who are interested only in the interface developed with FranTk may wish simply to skip to Section 2.4.4 and skim through that section.

2.4.1. Introduction to Air Traffic Control

Current ATC systems have a good safety record, but they are reaching their limits. Since 1987 the demand for air transport has been expanding at roughly 6% per year. EUROCONTROL - the European Organisation for the Safety of Air navigation - forecast that before 2015 air traffic levels will have doubled compared with those experienced in 1997 [49].

> "Current concepts have inherent limitations and cannot meet either the forecast traffic increase or the users' changing business needs. Particular shortfalls are:
> - Rigid airspace divisions and route structures;
> - Limited real-time information exchange;
> - Reliance on increasingly congested radio communications;
> - A lack of integrated planning between Air Traffic Management, airports and airspace users;
> - An inability to exploit aircraft avionics capabilities.
>
> Traditional methods of increasing capacity by further sub-dividing airspace sectors have reached their viable limit in some airspace areas."

EUROCONTROL, the European Air Traffic Control Agency, aim to handle the problem with improvements in planning and organisation, and an increased level of automation. As one potential solution, it is envisaged that more air-ground communications will take place via digital data-links, allowing efficient communications between airborne and ground systems, and between adjacent sectors. Controllers would be far more reliant on technology with such a solution. This has important consequences for the design of human-system interaction.

2.4.2. Developing Air Traffic Control Systems

Air Traffic Controllers work in a complex collaborative environment. *En-Route* Air Traffic Control involves flights travelling at high altitude across a number of different airspace sectors. Controllers work in teams within each sector and must also negotiate with controllers from adjoining sectors. New systems must therefore properly support this collaborative activity. Systems that reduce situational awareness could have serious safety implications [122]. A human-centred approach to the development of new ATC systems is therefore vital. Early user involvement is also needed to guarantee that new designs really support and improve the efficiency of controllers' work. Air Traffic controllers operate on tight time scales. Systems that interfere with existing work practices, by slowing down controllers, could be dangerous, and will be unacceptable. Numerous research projects have ultimately been rejected by controllers as unusable [122].

To ensure safety and maintain controller confidence ATC systems must be reliable. However, a number of problems with new systems have shown that this can be difficult. For instance, in 1992, while in actual use, a new system in Canada faced significant problems. "The system crashed in tests and actual use, freezing radar screens, displaying false information and even showing jets flying backwards". [202]

Organisations such as the UK's National Air Traffic Services (NATS) use a number of approaches when developing new ATC systems. These include formal approaches such as task analysis, human factors guidelines and error analysis; and user oriented approaches involving distributed multi-user simulations which take several months to develop. Full simulations must be used to gain an understanding of how controllers react to new ATC systems. However, these are slow to develop and demand too many resources to perform several design iterations. Smaller scale, more rapid approaches to user interface prototyping are therefore also important.

Given the collaborative nature of Air Traffic Control, the effectiveness of a new interface depends not only on the functionality of the system, but on the way it alters existing work practices. Simulations must be as realistic as possible. Support for experimentation with several users is therefore important. Prototyping tools should therefore support distributed, real-time, concurrent interaction. Despite the need for realism, less authentic studies can still be useful as problems with a system will scale up and out. If a controller finds a feature of an interface clumsy in a small-scale test, we can be reasonably sure the same will be true in real use. This means that rapid approaches to multi-user interface development can be very useful.

The development environment presented in this paper, fits in well here. It allows rapid prototyping and formal verification, thereby supporting usability and safety analysis. The next section presents the prototype Air Traffic Control system that was produced.

2.4.3. Air Traffic Control Background

The prototype system that was developed is for *En-Route* Air Traffic Control. An aircraft will be controlled by a number of different Air Traffic Service Units (ATSU) during the course of its flight, and is sequentially under the control of several controllers within a single ATSU.

The work in each sector is split into two major roles: a planning and tactical control. The planning controller is responsible for co-ordinating aircraft entering the sector and handing off aircraft from the sector. The tactical controller is responsible for communicating with aircrew and maintaining aircraft separation. Controllers must therefore be aware of what their partner is doing. In the UK, sectors are paired together. This means that the controllers covering the paired sectors sit side by side, with tactical controllers immediately beside each other. This allows them to handle emergency situations that occur at sector boundaries.

Controllers rely on flight plans, pilot requests, requests from other sectors, current weather and traffic conditions to manage air traffic. To gain a 2-D representation of traffic positions, controllers use a radar track data block, displaying aircraft and accompanying details showing the callsign and altitude. Controllers use a number of different interfaces in different centres, but these usually include a mouse, which can be used to calculate ranges between different aircraft and points.

Controllers use paper flight strips to keep a record of flight information and instructions. These also provide a legal record of controller behaviour for use in accident investigations. Flight strips consists of a band of paper printed with flight information containing the airline, flight number and type of aircraft along with the authorised flight plan (speed, level, route). Controllers use flight strips both as a memory aid and a means of communication [122]. Strips are laid out on a strip board. They can be arranged in a number of ways, for instance, by time and way point. Controllers slide strips from left to right to highlight different conditions such as two planes in conflict. Controllers can work simultaneously on the same strip board, and refer to particular strips on the shared board.

Controllers and aircrew communicate using radiotelephony (R/T) for all explicit communications. This provides a rapid and natural means of communication, which is particularly important when passing urgent communications. All pilots on a given frequency can hear every transmission. This is known as the "party line". It allows pilots to build up a picture of traffic based on what they hear of other transmissions. However, the quality of radio communications can be poor. Though controllers may be sure a pilot has heard a clearance, they cannot be sure that the pilot has not misunderstood, especially when communicating in an unfamiliar language. The effect of the party line can also be confusing with pilots responding to messages meant for other aircraft. Radio communications are also extremely congested. Radio bandwidth is the limiting factor on further growth in many sectors [49]. This congestion could be reduced. For instance, valuable bandwidth is being spent on passing routine sector co-ordination and frequency transfer messages.

In the future, it is envisaged that more air-ground communications will take place via digital data-link. It is hoped this will allow efficient communications between both airborne and ground systems, and adjacent sectors.

A number of data link based services are being considered. Our prototype concentrates on providing a subset of these services:

- ATC Communication Management (ACM).
- Clearance and Information Communications (CIC).

The ATC Communication Management service provides automated assistance to controllers and aircrew when transferring between airspace sectors. This, in particular, supports the role of the planning controller.

The Clearance and Information Communications service provides support for clearance, request and information dialogues between aircrew and controllers. Clearance dialogues allow the tactical controller to send orders to pilots; request dialogues allow pilots to negotiate flight parameters, most often heading or flight level, to minimise the cost of the flight or to avoid bad weather. Information dialogues allow the controller and pilot to share other information. Flight parameters can, for instance, be automatically downlinked to avoid the need for controllers to use valuable bandwidth asking for them. The CIC service therefore supports, in particular, the role of the tactical controller.

2.4.4. The ATC Prototype

The ATC Prototype that we produced allows several controllers to work together. It supports up to two controllers, planning and tactical, in two adjacent sectors. This provides a reasonable simulation of the paired sector setup discussed in the previous section. The prototype is heavily based upon designs produced as part of the EUROCONTROL EATCHIP Phase II HMI Catalogue[48]. The prototype does not consider the aircrew's view. All aircraft are simulated by the system.

An individual controller's view can be seen in Figure 3. It shows how a controller would send a flight clearance message to an aircraft, telling it to climb to flight level 240.

The interface provides a radar map of the sector with aircraft positions shown as *blips*. These show the current location of an aircraft and its last three positions, giving a good idea of aircraft acceleration. Associated with each blip is a label known as a *datablock* showing the Aircraft Callsign, next sector (or last sector if the aircraft hasn't entered this sector yet) and current flight level. Datablocks can appear in different colours depending on the status of an aircraft. For instance, an aircraft under the control of a

sector appears in black, an aircraft co-ordinating entry to a sector appears in blue. These labels will also show data-link error messages, and will highlight values undergoing data-link co-ordination. For instance, aircraft BAW33 has been sent a clearance to change its flight level.

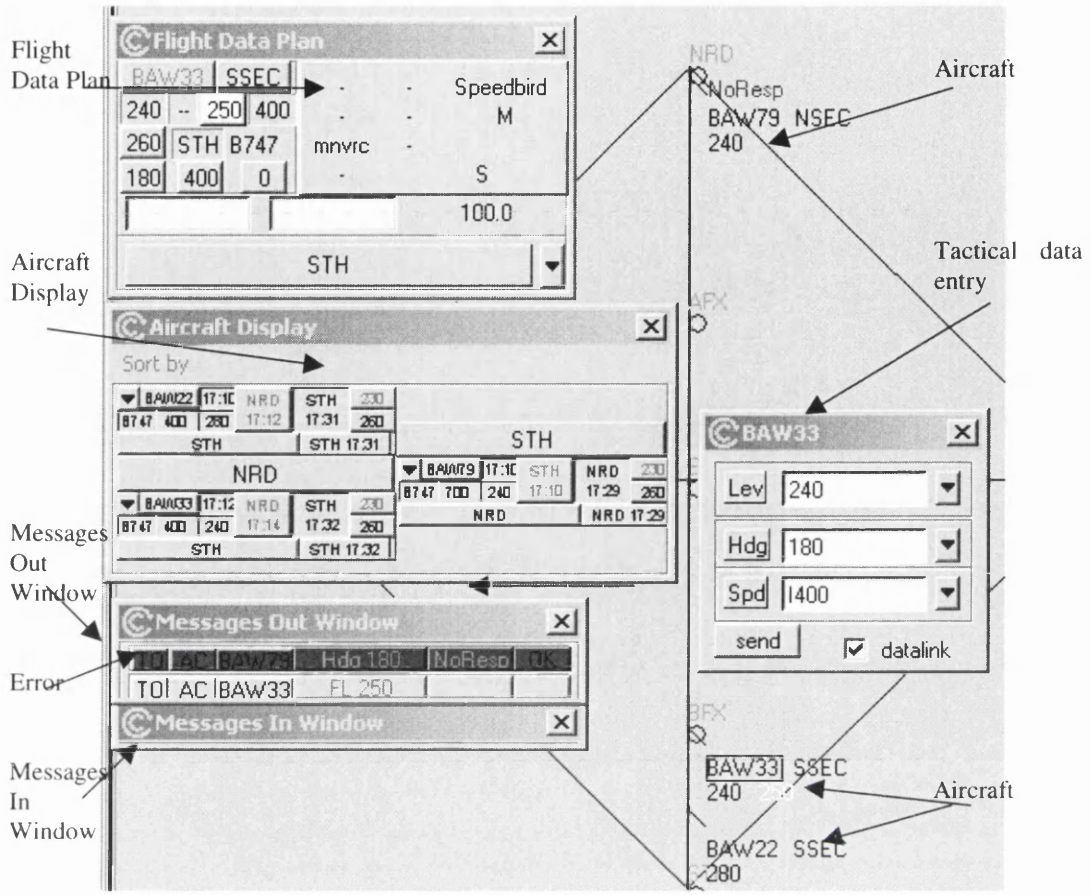

Figure 3 - A Controller's View in the ATC Prototype

By moving the mouse over a datablock, the controller can cause a *selected flight label* to appear. This shows more flight details, including downlinked flight parameters, and allows the controller to send flight clearance and co-ordination instructions. This direct provision of flight information reduces the need for radio communications between controllers and pilots. This allows controllers to keep their attention on the radar rather than being forced to move to the edge of the screen.

A more detailed *flight data plan* window is also available. It allows controllers to interact with the selected aircraft in a similar manner. It also shows more details, including downlinked controller preferences and the flight route. The plan shows information on the currently *hooked* (currently selected) aircraft, which will also be highlighted on the radar screen and on the Aircraft Display window.

Controllers can send data-link messages in a number of ways. They can send individual clearance messages. These can be immediate or conditional. For instance, Figure 3 shows the creation of a clearance to reach Flight level 240. This is done using the *tactical data entry* widget, which allows the rapid creation of composite messages, specifying heading, speed, and flight level. Controllers can send flight route updates with the graphical route editor shown in Figure 4. They can alter the heading and select and delete way points, seeing clearly the result of the change.

Co-ordination messages can be sent to the upstream sector (if accepting an aircraft), or to the downstream sector (if transferring an aircraft). Figure 5 shows a controller accepting a flight into their sector. They can either request the flight on a given frequency, or skip the flight telling it to pass to the next sector. The default frequency is shown, and can be set using a menu. Transfer of flights between

sectors can therefore be carried out at the press of a button. The appearance and behaviour of this co-ordination widget is modal and depends on the current status of the flight.

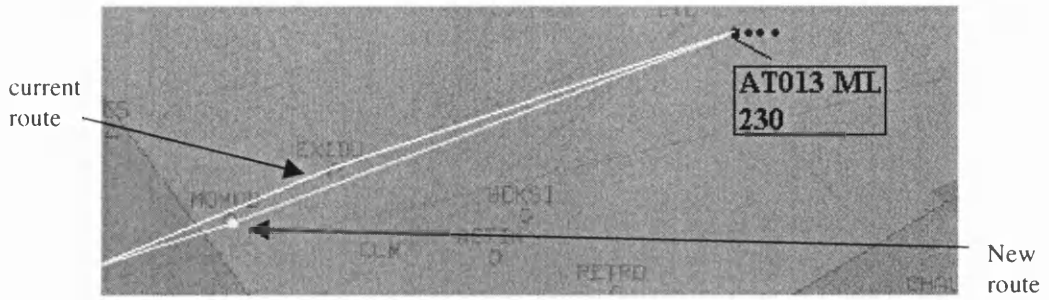

Figure 4 - Graphical Route Editor

Figure 5 - Sending a Co-ordination Message

Transfer parameters, such as the Transfer flight level, can also be negotiated, using a similar widget to that used to send a flight level clearance, by selecting the appropriate parameter on the *selected flight label.*

With any of these message creation approaches a data-link message can be sent or the ground system can be updated, by selecting the PHONE button. This can occur if a controller has been in radar contact with a pilot, or has spoken to a controller in an adjacent sector. This is important because some communication, particularly non-routine or urgent clearances will still be handled by radiotelephony. Data-link cannot provide the same tone of urgency as voice communication, and will be too slow for such messages.

Data-link communication messages appear in the *Message In* and *Message Out* windows. These allow controllers to keep track of incoming and outgoing messages. Both controllers within a sector can see the same message lists providing a shared view of data-link communication. Controllers can use the labels that appear in these windows to respond to messages. For instance, in Figure 3, an error message and a flight level message are both shown in the *Messages Out* window. Controllers can either reply that they are unable to co-operate with the request, or tell the pilot to standby, by pressing the UNBL and STDBY buttons respectively.

The Aircraft Display window can also be seen in Figure 3. It shows electronic flight strips for each aircraft. These can be highlighted by the controller, moving them from left to right for the controller, using the selection tags that appear on the sides of the strips. The Aircraft Display window can be ordered by flight level, entry time, exit time or organised by entry or exit point. In Figure 3, the Aircraft Display is organised by entry point. In this case the strips are organised into columns by entry point. Within each column they are arranged by time. This display is also shared between the planning and tactical controller, providing another means of maintaining a shared awareness.

2.4.5. Redesign

The initial interface underwent a process of interactive redesign, at the NATS headquarters, where I worked with a human factors specialist to change it to suit his needs. This process allowed us to test how well FranTk supported arbitrary modification of an existing application; whether it was powerful enough to support any change requested by the NATS specialist; and whether it was possible to develop a prototype of sufficient quality for the needs of an end user.

2.5. Summary

We therefore have three case studies of increasing complexity that were used to evaluate the scalability and applicability of the prototyping language, and the formal verification approach. The important contributions of each case study are summarised in the table below.

	Game	QOC Editor	ATC System
Scale	~300 lines of code	> 1500 lines	> 5000 lines
Real-time properties	Yes	No	Yes
Dynamic Displays	Yes	Yes	Yes
Need for Application/Interface separation	No	Yes	Yes
Multi-user	No	Yes	Yes
Scope for verification	No	Little	Yes
User testing and redesign	No	Yes	Yes

Table 1 - Summary of Case Study Contributions

Part II. Declarative Rapid Prototyping

Part II of this thesis presents FranTk, a new declarative language for developing interactive systems. It compares it with previous languages and demonstrates its benefits through the three case studies described in Chapter 2. This part is therefore aimed at readers interested in the design of FranTk and those readers interested in actually using it.

Chapter 3 – Declarative Development of Interactive Systems

A variety of languages exist to support the implementation of interactive systems. Some are based on specific architectures to allow a structured design approach. Others allow developers more freedom in how they structure a system. Some languages are visual and support development by direct manipulation; others are purely textual. All languages should support the development of three areas. They must support the development of the presentation or appearance of the interface; the dialogue level that describes how user interactions are to be interpreted, and the link between the interface and the underlying application.

In this chapter, I will review a number of these languages. When doing this I will consider a set of general requirements that are important to the development of the case studies discussed in the previous chapter. In particular, these languages must provide good support in the following areas.

1. The development of systems with both static interfaces, and dynamically changing interfaces.
2. The separation of application and interface code, so that one abstract model of the application can be maintained. To make this easier, some high-level mechanism should be available to ensure consistency between the application model and interface views.
3. The provision of temporal operators to support real-time interfaces. This includes the development of animations where the value of some parameter may change with time; and the specification of temporal predicates, such as time-outs, that are required by the Air Traffic Control case study.

I will first introduce a number of conceptual architectures, which attempt to support application/interface separation. I will then introduce the general areas of constraint based programming, model based programming and visual programming. I will then discuss in more detail a set of languages for the development of interactive systems. These include one object oriented language, Java, and a number of existing functional Graphical User Interface languages. Each of these languages will be illustrated by at least one example, the development of a "counter", with an increment and decrement button, and a label showing its current state. Finally, based on the general requirements outlined above, and on issues raised in the discussion, I will present a set of high level requirements for declarative implementations of interactive systems.

3.1. Conceptual Architectures

A number of conceptual architectures exist to help consider how the presentation, dialogue and application should be connected. The earliest of these, the *Seeheim* model[161] simply defined these three layers. It suggested that each layer should be considered as a separate component, and that communication could then take place between them, as shown below.

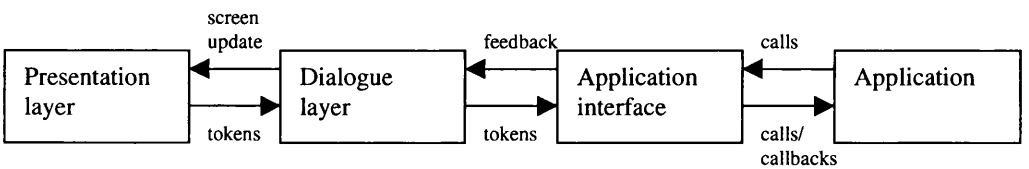

Figure 6 - Seeheim Model

It therefore provides separation between the application and interface. However, as each layer is monolithic, systems cannot be considered in terms of components. A number of approaches that allow design in terms of components, while still preserving application/interface separation, have been developed more recently.

3.1.1. The MVC Model

The Model-View-Controller (MVC) methodology has proved very influential. It was first associated with SmallTalk, as part of an object-oriented approach to design[70]. In MVC, a system is built using components. Each component is made up of three parts. The *model* handles application data. A model will contain an abstract representation of the data in a system. The model connects to a *view* that

maintains the appearance of the component, and the *controller,* which handles user input. These components can be combined into a hierarchy, as views can have subviews, which are themselves part of a component.

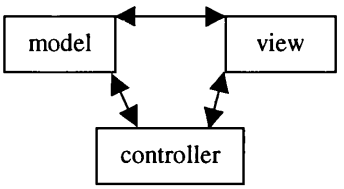

Figure 7 - MVC Model

3.1.2. The PAC Model

The Presentation-Abstraction-Control approach[32] below is similar to the MVC model. PAC programs are organised into hierarchies of components. The *abstraction* is similar to the MVC *model.* The *presentation* implements both the appearance of the component, and its interactive behaviour. The *control* maintains consistency between the abstraction and presentation. The tree hierarchy allows complex structures to be built up. Communication between different PAC components can take place through the control components.

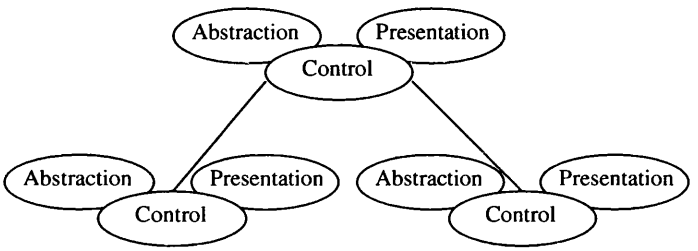

Figure 8 - PAC Model

Both the PAC and MVC models therefore allow systems to be thought of in terms of components. In MVC it is less clear how communication should take place between different components. The PAC tree makes it clear that this is handled through the *control* elements. Any MVC implementation must answer this question.

3.1.3. The ALV Model

Another similar model is the Abstraction-Link-View (ALV) model[90] below show in Figure 9. ALV programs are divided into two structures, the *abstraction* and the *view.* The abstraction covers the application model; the view covers the interface appearance and updates. Each of these parts forms a hierarchy. To maintain consistency between these two trees, constraints can be used. For instance, we could specify that the text of a label showed some value from an abstraction component. These ALV constraints are known as *links* and may be connected in any way between components in each tree. The ALV model has been used as the basis of the RendezVous language[91].

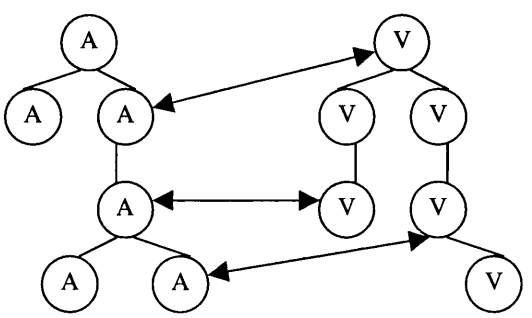

Figure 9 - ALV Model

The use of constraints here, can be very powerful. Rather than having to explicitly worry about sending updates between the abstraction and view, we can simply define a relation. This allows us to specify the relationship between the two in a declarative way. However, given a complex enough system, a miriad of constraints, going up and down the tree, could be difficult to understand and debug.

The application interface separation, provided by the models above, supports modularity in a design. This separation could perhaps allow designers to produce several interfaces to the same underlying application.

3.2. Constraints & User interface languages

Constraints can be used as a way of configuring interactive systems. They are fundamentally a declarative formalism, in that the programmer thinks about what conditions need to exist, rather than how they should be implemented.

They can also be used as a way of specifying the presentation of an interface. Geometric constraints can be used to define layout rules between objects on a screen. We could, for instance, produce a box and some text, and define the box to always surround the text. A variety of different interface languages use constraints. The RendezVous language, discussed in the previous section, is one example.

Probably the best known system that uses constraints is Garnet[136]. It uses an approach based on the MVC model to consider how systems should be structured. Constraints are used to maintain consistency between MVC components. Garnet allows the use of *one-way* constraints. These operate in only one direction. In the box and text example described above, though the size of the box is guaranteed to increase if the amount of text increases, if the user were able to increase the size of the box manually, then the size of the text would not increase. Garnet's successor, Amulet, supports *Multi-way* constraints. In our example, these would guarantee that the size of the text would increase with that of the box. They can, however, quickly become very complex to implement and use[136]. Garnet allows constraints to be used to maintain consistency between any data in a system, not just to prescribe the appearance of the interface. This provides for a very powerful system.

Garnet and RendezVous support indirection in constraints. This means that, rather than having to explicitly define a constraint between two components, we can instead attach a constraint to a pointer variable. The constraint can then be set to point to different objects at run time. This supports the definition of reusable modular components. It also adds expressiveness. We could, for instance, constrain a value to be based on a dynamically selected object[91].

However, the constraints available in Garnet can have serious problems[71]. Garnet permits two sorts of behaviour which can result in systems that are difficult to debug: side-effects and constraint loops.

Side-effects allow other actions to be performed when a value is updated via a constraint. For instance, Garnet uses constraint side-effects to dynamically create and destroy objects. This use of side-effects, means that programmers must know the order in which constraints will be evaluated.

Garnet also allows constraint loops. This means that two objects can have constraints that link back to each other. For instance, we could have two text boxes where the text in one box was constrained to be the same as the text in the other. We would therefore have an infinite loop. When combined with side-effecting this can be particularly dangerous. We could imagine an object that as a side-effect creates an instance of itself. This would result in infinite constraint loop, where instances of that object were continually created. This form of looping cannot be detected by a compiler. Sometimes it may even be undetectable at runtime[71]. These features can therefore make constraints very difficult to use.

3.3. Model Based Approaches to Interface Development

Section 1.3.1 introduced the model based approach to design. Recall that such systems attempt to derive the interface design from a selection of models. In addition to proof of concept prototypes, such as Adept[103], there are a number of systems that allow real interfaces to be developed. One of these is MOBI-D [162]. It allows more user control in the generation of an interface by allowing designers to

set style preferences that describe the type of interface to be produced. It also allows designers to override the system choices so as to provide direct control over the type of interface to be produced.

These approaches are, however, restrictive. Though they allow the generation of an interface from a model, or set of models, they do not allow the reverse. If problems are found in the resulting interface, changes must be made in the original high-level models. The mappings between these models can be complex enough to make this task difficult. The Teallach approach to model based design [78] attempts to avoid some of these problems by allowing transformation in any direction. It uses a mixture of models including a task model, a domain model, which defines the relationship with the underlying application, in this case a database system, and a presentation model (which defines the surface level issues of the interface). The Teallach project aims to allow partial translation between any of these models, so that a designer can start by considering the appearance of an interface before going back to a task model.

The automatic generation of interfaces is restricted to specific well-understood areas, where innovative design in an interface is not required. The Teallach approach, for instance, was used only to develop fairly standard interfaces to databases. Automated tools will generate reasonable, but unoriginal interfaces for a given system. If we wish to produce a new interface that attempts to better support a user's activities then we need to involve creative design. The creation of safety critical interactive systems falls into this category. Here a range of interfaces need to be developed, with direct expert involvement. The rationale for such designs must be clear, and responsibility for them must be explicit. Automatic model based approaches to design will not be helpful here.

3.4. Visual Approaches to Interface Development

Visual approaches, that allow interfaces to be built by direct manipulation, represent another popular declarative approach. Programmers can again define what an interface should look like, rather than saying how it should be produced.

The MEAD system [13] provides support for the construction of multi-user interface prototypes. It was developed to allow novel Air Traffic Control interfaces to be developed without the need for any programming. Interfaces are defined using three distinct sets of tools. An "Object Browser" is used to create simple data types with attributes and unique identifiers. Instances of these objects can also be created with given attribute values. Different views of this data can be created using the "View Definition Tool". These views can then be used to create "User Displays". The User Display definition tools allow objects to be displayed based on selection criteria (simple predicates). Presentation criteria are used to define which type of view should be used for any given object based on its attributes. Finally composition criteria are used to arrange collections of objects on a display. MEAD allows the development of fairly complex views. However, it does not allow these views to be defined as arbitrary functions of the application state, such as the creation of a line with a height constrained as a function of one or more object's attributes. More significantly, MEAD provides only very simple support for user interaction. Attributes can be edited textually via their views in user displays. The development of any more complex user interaction, such as the use of buttons, sliders and popup-menus is therefore impossible. Though MEAD proved useful for experimenting with possible ATC displays, it would be no use for the creation of complex interactive interfaces such as those in the data-link ATC case study used in this thesis.

Hypercard provides a good example of a visual approach for building interactive systems [9]. The programmer builds an interface from a selection of predefined objects, such as buttons, text fields and drawing tools. This allows designers to see what an interface should look like immediately. To define more complex properties, the programmer can open up a property sheet. This is used by the programmer to link the object to feedback, or to computation.

This direct manipulation style has problems. Certain things are difficult to define. For instance, while it is easy to build a static interface, defining an interface that can undergo complex dynamic changes is more difficult. Expressing complex layout rules, for example, to visualise graph structures can be impossible [71].

Environments that combine visual and textual programming have been developed in an attempt to combine the advantages of both. One of the most popular commercial systems is Microsoft's Visual Basic [129]. Visual Basic allows interfaces to be drawn using a predefined toolkit of components. The attributes of these components can be edited using a property editor. These components are defined as objects. To make these objects interactive, event handlers can be associated with them. These event handlers are defined textually. They update application code or update the display by changing objects' attributes. The relative simplicity of Visual Basic makes it popular for novice programmers. However, implementing complex programmes, particularly involving dynamically changing displays can be difficult. Visual Basic provides poor support for understanding the structure of a large program, which makes it difficult to incrementally modify such systems.

Some programming environments attempt to provide more structure, and allow the development of more complex interactive systems by combining visual and constraint-based programming. Garnet, for example, contains a tool called Lapidary [136]. This again allows designers to draw and define objects. These objects can be linked to constraints. Lapidary allows programming by demonstration: the designer can draw objects in different states, and Lapidary calculates the constraints to transform from one to another. Eventually, however, the programmer must use textual programming approaches.

3.5. Java's Swing – An Object Oriented Approach

There are a number of different object oriented languages that provide support for interactive system development. One of the most recent and popular is Java's Swing[194]. It supports both visual interface programming and a model-view-controller programming style. In this section, I will briefly outline the benefits and limitations of the support provided by Java.

JavaBeans [191] technology, of which Swing is a part, is designed to allow structured component based programming to be used in combination with visual interface builders. JavaBean components are known as Beans. Beans expose their features to builder tools by adhering to specific design patterns, and by using the Java Reflection API. This API allows tools to ask a component what methods it supports at run-time. Beans use *events* to communicate with other Beans. A Bean that wants to receive events (a listener Bean) registers its interest with the Bean that fires the event (a source Bean). Builder tools can examine a Bean and determine which events that Bean can fire (send) and which it can handle (receive). Beans can be composed visually using a builder tool. They can be composed geometrically into more complex displays; and they can be composed semantically by connecting listeners to events. However, to provide any actual behavior Java code must be written.

As well as supporting simple properties and methods, Beans also support *Bound properties*. Simple properties just support basic get and set methods. However, sometimes a component needs to be notified when a component changes. Whenever a *bound property* changes, notification of the change is sent to interested listeners. A Bean containing a bound property must maintain a list of property change listeners, and alert those listeners when the bound property changes. This provides a simple but powerful method of maintaining consistency between components that is an alternative to the use of constraint based approaches. Java's approach, however, has its limitations. Even using the *propertyChange* support provided, it would still require a reasonable amount of code to define a new bound property that was a function of several other bound properties.

Java's Swing provides a separation between user interface components and models which provide an abstract representation of some data. For instance, when implementing the "Counter" we could use a `BoundedRangeModel` to represent the application state. This supports methods to *set* and *get* an integer value, and to add a listener to hear about changes. To implement the example, we would therefore create the label and two buttons. We add a `ChangeListener` to the model that sets the value of the counter. We also add an `ActionListener` to each of the buttons, which sets the model to the appropriate value. The necessary code is shown below.

```
public class Counter extends JPanel
    implements ChangeListener,ActionListener {

  public Counter(BoundedRangeModel state) {
    lbl = new JLabel("Counter:0");
```

```
    inc = new JButton("inc"); dec = new JButton("dec");
    this.state=state;
    state.addChangeListener(this);
    inc.addActionListener(this); dec.addActionListener(this);

    // Layout components
    setLayout(new GridLayout(2,1));
    add(lbl);
    JPanel tmp = new JPanel(new FlowLayout());
    add(tmp);      // place the two buttons in a panel of their own
    tmp.add(inc); // to enable them to be side by side,
    tmp.add(dec); // below the label
  }

  JButton inc,dec;
  JLabel lbl;
  BoundedRangeModel state;

  // Implement ActionListener,
  // This action is performed on button clicks
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==inc) // If it is the inc button, add 1
        state.setValue(state.getValue()+1);
    else if (e.getSource()==dec) // with the dec button, subtract 1
        state.setValue(state.getValue()-1);
  }

  // Implement ChangeListener,
  // Set label when state of model changes
  public void stateChanged(ChangeEvent e) {
    lbl.setText("Counter:"+state.getValue());
  }
}
```

The layout combinators used are slightly clumsy. We need to create a subpanel in which to place the two buttons. Layout would have been easier if we could provide a simple declarative definition of the appearance of the interface: `above(lbl,beside(inc,dec))`. While this use of layout combinators can be avoided when building interfaces visually, it is still necessary when constructing dynamic interfaces.

The use of abstract models is very important, as it satisfies the requirement of supporting application/interface separation. There are a range of other models provided by Swing. For instance, there is a `ListModel` which defines the methods which components such as listboxes use to access lists. Though Java provides `ListModels`, they are less powerful than standard lists. For instance, it would take a fair amount of code to define a `ListModel` that was always sorted. Life also becomes more difficult when modelling new data types. In this case, programmers would have to define a new model class.

As we have seen, user interface components and models communicate via listeners. Java distinguishes listeners based on what they can be added to, rather than simply on the type of information that they consume. This means that to consume values of new types we must define new Listener classes. It is also sometimes difficult to add a listener in two places. For instance, Java distinguishes between `MouseListener's` which hear about mouse clicks, and `ActionListener's` which hear about action events (such as from buttons); yet both can be thought of simply as actions that are uninterested in the data that they consume . Here we would need to duplicate code.

To support new data types it is necessary to define new kinds of events. However, doing so is tedious, because the Java's `AWTEventMulticaster` class (which manages listener lists) only supplies a fixed set of overloadings for the listener `add` and `remove` methods. New kinds of events may easily have signatures that do not match any of the given overloadings. In such a case, the programmer of the new kind of event must also implement all of the list management needed to support multiple listeners.

Conversion between listeners is also cumbersome. For instance, if we had an item listener (which hears about selected objects), and wanted it to be fired every time a button was pressed, we would require to write the code below. It creates a new `ActionListener`, which fires the `ItemChange` listener.

```
ActionListener l = new ActionListener () {
 public void actionPerformed(ActionEvent e)
 {i.itemStateChanged (new ItemEvent (e.getSource(),
                                     ItemEvent.ITEM_STATE_CHANGED,
                                     obj, ItemEvent.SELECTED))
}};
```

In summary, Java's Swing has a number of important features.
1. Through JavaBeans it supports visual construction of static interfaces
2. It supports application/interface separation via models.
3. It allows consistency to be maintained between models and views via listeners.

However, Java also has some limitations.
1. The way layout is handled is very imperative, making it less succinct than declarative alternatives.
2. Models and Listeners have been developed to handle very specific types of data. They have a range of methods which make them useful for specific purposes, such as representing the state of a slider, but not for representing values of any given type. The mechanisms necessary to create new types of Model and Listener are relatively cumbersome. Converting between different types of listener is also cumbersome.

3.6. Functional Approaches

The systems mentioned above though relying on declarative approaches, such as constraints, are all based in imperative languages. They support potentially dangerous concepts such as side-effects, which can make life difficult for programmers. An alternative approach is to use truly declarative languages to implement systems. These can give designers greater faith in the correctness of their programs. They are also potentially easier to relate back to design notations [3]. Functional programming provides one such purely declarative framework. This sections discusses some functional approaches to interactive systems development.

In particular, this section will concentrate on the use of Haskell [158]. Haskell is a purely functional language that supports a number of useful features, including higher order functions, static polymorphic typing, a lazy semantics, rich data types and a monadic I/O system.

Haskell has been the focus of a lot of recent work on user interface toolkits. There have been four main approaches used to structure user interface code in functional programming languages, and in Haskell in particular.

- **Callbacks** – Systems such as TkGofer use a simple callback based approach to programming.
- **Stream processing** – User interface components can be viewed as stream processors, that consume streams of user input and produce streams of output commands. Fudgets and Gadgets are two systems that take this approach.
- **Imperative concurrency** – User interface components can be structured as a set of processes that execute concurrently and consume user input. Haggis is a good example of such a system.
- **Constraint based approaches** - We can introduce the notion of some form of reactive behavioral value that can change value over time. We can then view an interface as function of some set of values. Clock, Pidgets and Fran have all introduced such concepts.

I will first introduce the basic concept of performing I/O in Haskell, before going on to discuss these four approaches to interactive system development.

3.7. Performing I/O in Haskell

Haskell uses monadic IO to support sequencing of actions. Older approaches to sequencing used stream based I/O, which produced confusing code. For instance, the following simple program, copies its standard input to its standard output [70]:

```
main ~(Str input: ~(Success : _)) =
  [ ReadChan stdin,
    AppendChan stdout input]
```

How input, is transferred between `ReadChan` and `AppendChan` is unclear. The resulting confusion caused serious problems for programmers learning to exploit functional programming.

Newer approaches based around monadic I/O provide a more imperative programming style, familiar to most programmers [156]. The same program, with monadic I/O, would be:

```
main = do
         ch <- getChar
         putChar ch
         main

getChar :: IO Char
putChar :: Char -> IO ()
```

The last two lines are examples of Haskell type signatures. In Haskell the syntax '`::`' is used to denote that a value has a given type; the argument types are then separated by '`->`'; in type declarations, non-capitalised names, like `a` here, are *type variables*, indicating polymorphism, i.e., the ability to work with all types; application of a function '`f`' to arguments '`x , y, ...`' is written simply '`f x y ...`'. The first type signature, for instance, says that there is a function called '`getChar`' who's type is '`IO Char`'. The second says that '`putChar`' is a function which takes one argument of type '`Char`' and returns an a value of type '`IO ()`'.

The type `IO a` can be thought of as an I/O action that returns a value of type `a`; the type `IO ()` is therefore an action which when performed will do some computation and return no useful value. Here we can think of `()` as the C or Java `void` type. Here `getChar` is therefore an `IO` action that returns the character read, and `putChar` is a function that takes a character and performs an `IO` action that prints the value and returns no result. The sequencing of actions is more explicit here; the input from `getChar` – `ch`- is used by `putChar`.

The monadic expression above is special syntax for the standard monadic combinators.

```
class Monad m where
 (>>=) :: m a -> (a -> m b) -> m b
 return :: a -> m a
 (>>) :: m a -> m b -> m b

 a >> b = a >>= \_ -> b

instance Monad IO
```

Here *Monad* is a *type class*. This defines a general interface that may be implemented for a number of different data types. This allows ad-hoc overloading in Haskell.

A monad is a family of types '`m a`' based on a polymorphic type constructor '`m`', with functions `return`, `(>>=)` and `(>>)`. Here the function `(>>=)` sequences two monadic actions, using the result of the first in the second action. The function `return` simply returns its argument value without any additional computation. Finally the function `(>>)` acts like `(>>=)` except that the value returned by the first argument is discarded, rather than being passed to the second argument. The expression '`\_ -> b`' is a *Lambda expression*, which denotes a function. After a '`\`' we list the arguments of the function, then a '`->`' and then the result. We can therefore define the two expressions below in terms of these monadic combinators.

```
do {x <- act;f x} == act >>= \x -> f x
do {act1;act2} == act1 >> act2
```

Readers who require further explanation are referred to [201] or [94].

3.8. Functional Callback based approaches

User interfaces can be constructed in terms of callbacks. When creating a component, we associate an action with it that will be performed when it hears some user input. For instance, we could define a *push-button* using the following code:

```
button :: String -> IO () -> IO Button
```

Here `button` is an IO action that that takes two arguments, a String label to display and an action to perform when clicked. When performed it produces a value of type `Button` that represents a handle through which the object can be accessed. We can now imagine a number of operations on these button handles to change their appearance, such as setting the colour or changing the label.

```
setLabel :: Button -> String -> IO ()
```

The button's callback could therefore be set to perform one of these output actions.

3.8.1. TkGofer

The TkGofer toolkit [204] uses a callback based approach. TkGofer uses an interface to the popular Tcl/Tk scripting language to provide a platform independent set of widgets. The layout and behavior of components are described in Gofer (a variant of Haskell). TkGofer makes use of mutable variables to allow interface components to have state. To create our example "Counter" with two buttons, up and down, we would use the following code:

```
main :: IO ()
main = start updown

updown :: GUI ()
updown = do st <- newState 0
            win <- mkWindow [title "up-down counter"]
            lbl <- mkLabel win [text "0"]

            let action :: (Int -> Int) -> IO ()
                action f = do
                 val <- readState v
                 let val' = f v
                 writeState v val'
                 cset lbl [text (show val')]

            incb <- mkButton [text "inc",command (action inc)]
            decb <- mkButton [text "dec",command (action dec)]
            pack (above lbl (beside incb decb))
```

We create a mutable variable using `newState`, to store the state of the "counter". We can then use and update the value of that variable later. The GUI monad is built on top of the IO monad and enables Tcl-Tk actions to be carried out.

```
newState :: a -> GUI (State a)
readState :: State a -> GUI a
writeState :: State a -> a -> GUI ()
```

We then create a window, and within it we create a label and two buttons.

```
mkWindow :: [Conf Window] -> GUI Window
mkLabel :: [Conf Label] -> GUI Label
mkButton :: [Conf Button] -> GUI Button
```

When creating widgets we pass in a list of configuration information. For instance, we pass the label some text to display. Configuration information has a type associated with it, so only the correct options can be passed to a given widget. The type, `Conf Label`, represents any configuration information

that is valid for a label. TkGofer uses type classes to restrict which configuration information can be passed to which component. For instance, there is a `Has_Text` class. It has one method, the text method which takes a string and produces a configuration option. The `text` option can only be used with instances of the `Has_Text` class. These include `Buttons` and `Labels`, both components that can display textual labels. (The constraint `Widget w`, means that components that can take `text` configuration options must all be instances of the `Widget` class.)

```
class Widget w => Has_Text w where
  text :: String -> Conf w

instance Has_Text Button
instance Has_Text Label
```

We compose widgets geometrically using combinators such as `above` and `beside`. These take two widgets and return a `Frame` widget. We then `pack` the created widget on to the screen.

```
above,beside :: (Widget w1,Widget w2) => w1 -> w2 -> Frame

pack :: (Widget w) => w -> GUI ()
```

Finally, we define the behavior of the buttons. We do this using a `command` callback. Each button first updates the state, by applying some function to the current value (either incrementing or decrementing). It then resets the text on the label.

```
class Widget w => Has_Command w where
  command :: GUI () -> Conf w

instance Has_Command Button

cset :: Widget w => w -> [Conf w] -> GUI ()
```

3.8.2. Discussion

The approach taken in TkGofer has a number of positive features.

- Typed configuration options – The use of configuration options makes it easy to create a component which may also have a range of default values. The use of type classes allows us to easily constrain which options can be given to which component.
- Functional layout combinators – Widgets can be geometrically composed with a range of functional combinators, making it easy to generate complex displays.

However, TkGofer also suffers from two major constraints.

- Grouping components – While we can compose pairs of components, we can't apply these composition operators to collections of components. For instance, it might be useful to have a combinator that places a list of components above each other. This is because different widgets have different types, so we could not form a heterogeneous list consisting of buttons and labels. Instead, we need to explicitly coerce each widget into a Frame and then compose them.

  ```
  hbox :: [Frame] -> Frame
  frame :: Widget w => w -> Frame
  composite = hbox [frame lab,frame btn]
  ```

  This approach is less than ideal.

- Spaghetti of callbacks – Relying on callbacks can make TkGofer programs difficult to structure. The user interface takes control from the application. In particular, components need to maintain references to each other in order to perform their tasks. In our example, each button needs a reference to the label and must update both the application state in the mutable variable and the label's display itself. When programming large systems these references turn the program into a

'spaghetti of callback' [137], making the structure difficult to understand. More powerful mechanisms can be built on top of TkGofer to help overcome this problem, such as introducing an MVC style of programming [27]. Here components update a model, and widgets register interest in the model to display its value.

3.9. Stream processing - Fudgets

The stream processing approach to programming considers a user interface to be a stream processor that consumes user input and produces output events. We will consider two variants of this approach, Fudgets and Gadgets.

3.9.1. Fudgets

The Fudgets system [25] was the originator of the stream processing approach. An interface is described in terms of a series of components or *Fudgets*. Each Fudget receives data on input streams and sends it out on output streams. Fudgets can be combined to form composite Fudgets using a set of combinators.

Fudgets can be connected together. For instance, we can connect the output stream from one fudget to the input stream of a second using (>==<). To allow a Fudget to listen to two other Fudgets, we must compose them together (using >+<). This forms a new Fudget that produces messages of a sum type. Values will either be `Left a`, if they come from the left hand fudget, or `Right a`, if they come from the right hand fudget. This tagging can become awkward when forming large Fudgets. To overcome it we can sometimes generate Fudgets that send streams of the same type. For instance, in our example "Counter", we could create button Fudgets that send integer modifying functions. These can then be composed without the need for tagging. The example "Counter" can be implemented in Fudgets as follows:[2]

```
import Fudgets

main = fudlogue (shellF "Up/Down Counter" updown)

updown = intHolderF >==< buttonsF

buttonsF = buttonUpF >*< buttonDownF
 where buttonUpF = buttonMsgF increment "up"
       buttonDownF = buttonMsgF decrement "down"

intHolderF :: F (Int -> Int) a
intHolderF = intDispF >=^^< stateHolderSP 0

stateHolderSP :: a -> SP (a -> a) a
stateHolderSP s = mapstateSP hold s
  where
   hold s f = let s' = f s in (s',[s'])

buttonMsgF :: m -> String -> F Click m
buttonMsgF m s = tomsg >^=< buttonF s
  where tomsg Click = m
```

We create two button Fudgets that transform their clicks into update functions, using the message mapping combinator >^=<. We then compose the two Fudgets with >*< to produce a composite untagged fudget. Next we connect this composite fudget to a state holder that maintains the current value of the fudget. Finally, we connect this state holder to an integer display fudget that displays labels.

The Fudgets programming model combines the notions of semantic and geometric composition. We compose Fudgets with respect to their input and output streams. They are then given a default layout. If

[2] This example is taken from [144]

we need a display that is separate from the semantic composition, we must explicitly name Fudgets in order to apply alternative layout combinators.

The basic model, demands that each fudget can read from only one, and write to only one other fudget. This makes it difficult for a component to inform several other components of a state change. There is no high level support for maintaining consistency between several Fudgets. Sharing data between Fudgets is also impossible. All consistency conditions and data communication must be individually programmed [71].

To summarise Fudgets allows applications to be composed in a functional style. However, it suffers from a number of problems as a result:

- Need for explicit tagging when composing Fudgets. The use of one output stream makes it hard to see where a fudget is sending its output values [62].
- Combination of semantic and geometric composition, makes it more difficult to produce complex displays where the two concepts are not related.
- Parameterising a Fudget over the type of elements transmitted makes it difficult to provide combinators to combine collections of components.
- Because of the static structure, it is also difficult to program systems with dynamically created and deleted objects [144].

3.9.2. Gadgets

The Gadgets system developed by Rob Noble [144] attempts to overcome the problems visible in Fudgets by introducing explicit channels. A user interface is considered to be a set of components, or *Gadgets*, that communicate on wires, channels with an input end and output end. We therefore define a button as a Gadget, which displays a label, and takes a value, which it emits on a given output port, every time the user clicks the button.

```
button :: String -> a -> Out a -> Gadget
```

The Gadget system introduces the notion of a process to allow components to individually perform calculations and communicate on wires. When a Gadget hears a value on a wire it must itself become runnable and perform some communication. When one Gadget waits on some input from a user, it must not block every other Gadget. We can create new processes using the spawn primitive:

```
spawn :: ComponentClass b => Process a -> Process b -> Process b
```

We can implement the example "Counter" as follows[3]:

```
counter :: Gadget
counter =
  wire $ \w1 ->
  wire $ \w2 ->
  let incbtn = button "Inc" (out w1) increment
      decbtn = button "Dec" (out w1) decrement
      lab = label "0" (in w2)
  in
  spawn (count 0 (in w1) (out w2)) (lab <|> (incbtn <-> decbtn))
        -- lay the two buttons side by side, under the label
  where
   count :: Int -> In (Int -> Int) -> Out String -> Gadget
   count n i o = rx [from i $ \f ->
                     let n' = f n in
                     tx o (show n') $
                     count n' i o]
```

We create two wires and four gadgets. The two buttons talk to the first wire, passing integer modifying functions. A third gadget is spawned which maintains the state of the application. It reads values from

[3] This example is taken from [62].

the first wire (using rx), and applies them to its current value before transmitting them on the second wire (using tx).

The use of wires in Gadgets is a powerful concept. We can create a component by passing in all the arguments necessary to generate it. This allows us to define a generic component type and so define composition functions that operate on collections of components. Each process can handle an element of the application code. The semantic wiring is no longer associated with the geometric composition making the structure of the program more modular.

However, Gadgets relies on the older continuation passing I/O style (see Section 3.7). The need for true concurrency makes non-determinism an important issue. It is also still difficult to form complex compositions of wires or to specify complex temporal constraints.

3.10. Imperative Concurrency - Haggis

The Haggis toolkit developed by Sigbjorn Finne [62] provides a good example of an imperative concurrency approach. An interactive system is defined as a number of concurrent threads. These communicate by message passing, and through shared data, implemented by a series of imperative commands.

Haggis makes use of Concurrent Haskell [157]. This language extension supports lightweight processes, and makes use of monadic I/O. Programmers can create new child processes with the `forkIO` function. Communication occurs asynchronously, through shared variables (MVars) which operate like semaphores.

Haggis has a number of key features, described below.

3.10.1. Virtual I/O

The use of concurrency provides for modular design. A common problem with many graphical user interface systems is their reliance on an event loop, and callbacks. This style has well known problems [137]. Haggis, instead, treats the user interface as a virtual device, allowing the application to maintain control. The concurrent features of Haggis allow several virtual I/O devices to operate at once. For instance, one process could block waiting for a mouse click while others go on with necessary work. It is this feature that supports the imperative concurrent style of programming.

3.10.2. Declarative structured graphics

A further problem with the development of graphical interfaces is that conventional languages tend to be highly imperative. This means that rather than considering how a picture should look, programmers must describe the sequence of actions that must be used to render it. This added complexity makes mistakes more likely. In contrast, in Haggis all static graphical output is specified declaratively through a Picture type [59]. Haggis provides operations to transform and combine objects, along with an extensive list of graphical primitives. For instance, we could specify the image shown in Figure 10 with the associated code.

```
enemy = fillSolid $ withColour grey $
        beside (ellipse (5,12))
               (above thruster thruster)
 where thruster = coverlay cross
                           (ellipse (15,6))
          -- centre one image over another
       cross = withColour black $
               beside (rectangle (12,2))
                      (rectangle (2,8))
```

Figure 10 - A Haggis Picture

Haggis takes care of converting pictures into calls to the window system, with the Glyph output abstraction. The glyph function takes a picture and produces an interaction object that displays it.

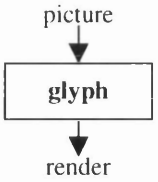

Figure 11 - A Haggis Glyph Component

This declarative approach makes it easy to build, and manipulate complex images. However, Haggis provides a separation between static pictures and interactive widgets such as glyphs. This makes it more difficult to describe complex interfaces where individual components may evolve dynamically.

3.10.3. User interface, application separation

Geometric and semantic composition are separated in Haggis. User interface components are represented by an object, from which we can extract a handle for visual composition. For instance, the glyph output abstraction returns a Glyph handle, which the application can use, for instance, to update the image. From this we can extract a Display Handle, which is a reference to the interactive graphical surface.

```
glyph :: Picture -> Component Glyph
getDH :: Glyph -> DisplayHandle
```

This separation allows programmers to build more modular systems. We can define the appearance of a system and its behaviour separately.

3.10.4. Compositional structure

Haggis provides layout combinators and other functions to combine the Display Handles mentioned above. For instance, we could make an interactive widget by combining an input event controller with the Glyph described above.

```
widget picture dc =  do
      gl <- glyph picture dc
      catchMouseEv gl
```

This produces a widget as shown in Figure 12.

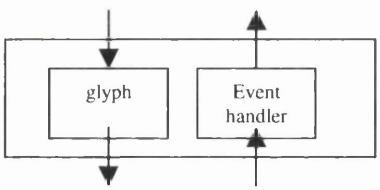

Figure 12 - A Haggis Interactive Widget

We can then compose these interactive objects to produce an interface as shown in Figure 13.

```
main = do
  dc <- mkDC []
  obj1 <- widget (text "press 1") dc
  obj2 <- widget (text "press 2") dc
  realiseDH dc (above obj1 obj2)
```

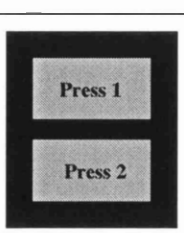

Figure 13 - A Simple Interface in Haggis

3.10.5. An Example

We can define the example "Counter" in Haggis as follows:

```
main = do
  dc <- mkDC []
  lbl <- label (show "0")
  btn1 <- button (text "inc") increment
  bnt2 <- button (text "dec") decrement
  btns <- hCombine [btn1,btn2]
  forkIO (interact lbl btns 0)
  return (vbox [getDH lbl,hbox [getDH btn1,getDH btn2]])
 where
   interact :: Label -> Button -> Int -> IO ()
   interact lbl btns n = do
     f <- hGet btns
     let n' = f n
     hSet lbl (show n')
     interact lbl btns n
```

We create two buttons, an increment and decrement button. We compose the input from these two buttons to form a composite button. The view consists of a label above both buttons. We then start off a concurrent process that maintains the value of the label. When it hears an update-function message resulting from a button click, it applies the function to its current value and displays it. The Haggis approach therefore works well for simple examples like this, providing good semantic separation between the view and the application.

3.10.6. Discussion

Dealing with dynamic interfaces, with varying collections of widgets on screen, is difficult in Haggis. Haggis differentiates between widgets and pictures. The graphical combinators available for widgets are much less powerful and less declarative. In particular, to handle collections of widgets we require to use the Composite Container component which has an imperative interface. Providing an abstract model of a collection of objects causes further problems. An application would be modelled as some mutable state. A process would then be required which updated the display every time this state was changed. To allow multiple views of this data we could have one application process which received inputs on a channel and sent out updates on a channel. However, as this solution would be non-deterministic there would be no guarantee that the set of interfaces components would all be up to date. Alternatively, each interface component could register update callbacks with the application. Doing this we would have to be careful not to cause a deadlock. Either mechanism is somewhat clumsy and error prone.

Concurrent communication in Haggis is based around shared semaphore variables. These provide a very primitive interface for shared communication. It is, however, possible to build more powerful communication mechanisms on top of these. Concurrent Haskell, for instance, also provides asynchronous channels for message passing. The range of temporal operators is still somewhat limited. For instance, to make one process suspend and resume another requires some fairly sophisticated programming. One possibility is to introduce a set of richer, synchronous communication primitives. Inspired by work modelling interactive systems as LOTOS processes [152], I developed a set of LOTOS like primitives for use with Concurrent Haskell and Haggis [170]. Using this new communication library I developed the Space Fighter Game, introduced in Chapter 2. The combination is discussed in Appendix C.

3.10.7. Summary

Haggis therefore provides a good powerful basic model to consider the design of interactive systems. However, it has a number of problems.

1. The approach exposes programmers to the perils of concurrent programming, such as non-determinism and synchronisation.
2. Processes can share data and so provide some form of data dependency between objects. However, defining constraints between different processes is impossible.

3. Pictures are static datatypes. Interactive components must be composed together through their *Display Handles*. The composition operators for *Display Handles* are less expressive. To build an interface with a number of dynamic components we must either use Display Handles, and suffer the lack of expressiveness in terms of image composition, or have a single process that accepts all events and then uses these to manipulate individual picture components.
4. While it provides separation between application handles and display handles, it does not explicitly support component based, application interface separation. This means it is up to the programmer to attempt to build systems in a modular way. In particular, it is difficult to provide multiple views of a dynamic interface.

3.11. Functional Constraint based approaches

There have been a number of attempts to develop functional languages based on constraint ideas. Three examples of this approach are Clock [71], Pidgets [178] and Functional Reactive Animation [44].

3.12. Clock

Clock has been designed as a constraint based functional language [71]. It has a graphical architecture language that can be used to describe how systems fit together and a textual language to describe the behaviour of each component. It is based on the MVC model.

Programmers first structure a system by decomposing the interface from a root view into a tree of views. This decomposition is based on a hierarchical display model of an interface. Such Clock architectures are produced interactively using a visual tool called *ClockWorks* [134]. This allows programmers to design and modify their architectures easily. It provides easy access to a library of components. Fast iterative design is therefore possible.

Each component in a Clock architecture tree contains an *event handler*, which takes user inputs and sends updates. This *event handler* (EH) is similar to the *controller* part of the MVC model. Components may also contain *request handlers* (RH) which represent the MVC *model* and can receive inputs and accept requests. Finally, a component has a view, which is defined as a relation of the model. This is therefore similar to the ALV models mentioned earlier. The behavior of each of these components is described textually.

We will consider three examples in this section, our "Counter" application, and the Space Fighter and QOC editor applications from Chapter 2.

3.12.1. The Counter

The architecture for the "Counter" is shown in Figure 14.

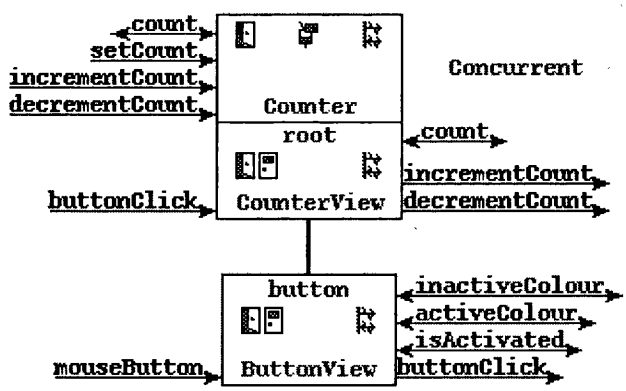

Figure 14 - The Clock Architecture for the "Counter"

The *Counter* component is a Request Handler. Counter has three methods, which we will use: *incrementCount, decrementCount* and *count*. The first two methods *update* the value of *Count* by incrementing and decrementing it, respectively. The last method *requests* the value of *Count*. The root component is of class *CounterView* and contains both the view and the controller of the component. The component accepts *press* events from its child buttons. It can use the *count*, *incrementCount* and *decrementCount* methods.

The controller part of the component performs increment and decrement actions depending on which button has been pressed.

```
buttonClick "inc" = incrementCount
buttonClick "dec" = decrementCount
```

The view part of the component is defined as follows:

```
view = above (numText count)
             (beside (button "inc") (button "dec"))
```

This view uses a constraint from the model. It uses two instances of the button subview. Subviews are identified by String names. These names are persistent. For instance, whenever a button subview with the name "inc" is referred to for the remainder of the program, it means the same instance. Any local state within the component will therefore persist. To create a new instance of a subview with new state, we would therefore need to define one with a different name. The view function simply specifies a relation between the model and the user interface appearance.

This use of String names to distinguish components is clumsy and error prone. For instance, spelling errors in component names will not be caught until runtime. It also makes it difficult to garbage collect a component as there is no way to tell if it will be used again later in the program.

The complete code for the *Count* RH is:

```
type Count = Num
setCount n = save n
decrementCount = save (this - 1)
incrementCount = save (this + 1)
count = this
initially = save 0
```

The predefined function *save*, sets the new state of the RH. The predefined function *this* returns the current state of the RH. Therefore the *count* method simply returns the current value held in the RH, and the *incrementCount* method sets the new value of the RH. The *initially* function says what to do when an RH is created. Though some of this may not look safe, the semantics of Clock are defined to guarantee referential transparency.

There is a strict set of rules defined to explain how these updates can be used. Updates can only travel up the architecture tree. They are guaranteed to terminate at or before the root. Infinite constraint loops are therefore impossible. Requests can also only travel up the tree, so a component can only use constraints based on values in its parent components' RHs. A component can use 0 or more instances of each of its subcomponents. These are created through the subview relationship.

The Clock architecture language provides good support for iterative design [71]. Requests and updates from a component are routed automatically by Clock. Programmers do not explicitly say how components are to be connected. The only explicitly defined relationship is the subview. Requests and updates are simply routed up the tree to the first component that can deal with that type of action. This means that it is easy to move objects around in the tree, as no explicit connections will be broken. For instance, if we wished an RH that had been used by one component to be shared by several, we could just move it up the tree. This would not cause any problems. However, this use of untargeted method calls can be unhelpful. For instance, it makes it impossible to have two instances of the same request handler in scope at the same time.

Clock also supports groupware applications [140]. View functions are provided to allow windows to be opened up on different users screens. A distributed version has also been developed. Programmers can define some parts of the tree to exist on the user's side while others exist on the client's side. This aids the development of groupware applications.

3.12.2. The Space Fighter Game

The first stage in producing an implementation in Clock is to produce the architecture. The architecture for the Space Fighter game is shown in Figure 15. The system is made up of the *ToolBar* and the *Screen.* The screen decomposes into *ShipViews*, *LaserViews* and *EnemyViews*.

We next add the RHs necessary to implement the system. We use a *Paused* RH, which defines whether the game is paused or not. We have a *GameState* RH which maintains the state of the lasers, enemies and ship. The RHs are then placed in the tree at the lowest level possible, so that they are visible to all components that use them.

The graphical architecture makes it easy to see how the system is composed together. The separation into RHs and control components affords good application/interface separation. In this example, however, there is little need for any great separation because there is only one view of the game.

3.12.2.1. The Game State Component

The GameState RH is associated root component. It consists of a list of lasers, enemies and ship data. There is also a Boolean which is True if the game is over, and a current score.

```
type GameState = ([Laser],[Enemy],Ship,Score,Bool)
```

It has four update methods: ship *move* updates, *timer* updates; *restart* updates and *quit* updates, which manually set the state to '`game over`'.

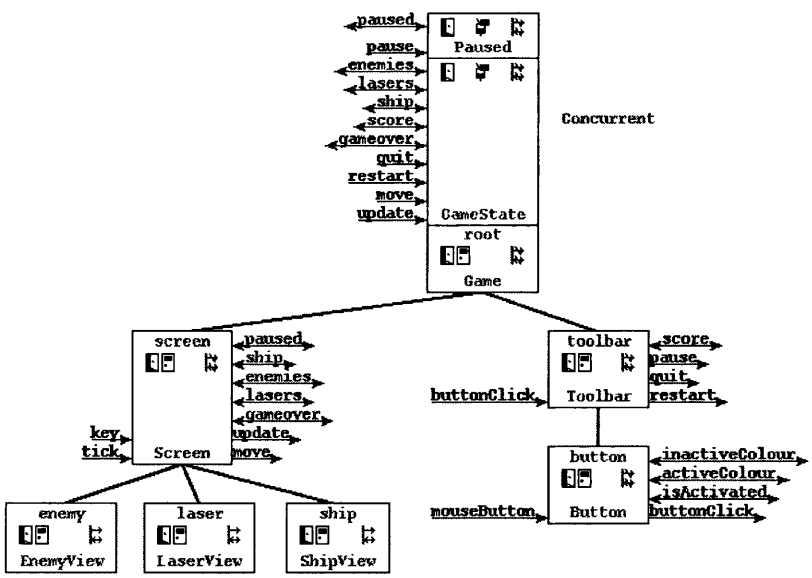

Figure 15 - The Clock Architecture for Space Fighter Game

On each *timer* update the component moves the enemies and lasers along the screen. This can only happen when the game is not paused or over. Once the lasers and ships have been moved, we calculate any overlaps, deleting any enemies and lasers that overlap, and ending the game if the ship overlaps with an enemy.

```
timer =  save (collide (map (moveObj Right) lasers,
                        map (moveObj Left) enemies,
                        ship))
```

When it receives move updates, it moves the ship in the required direction. It then calculates if the ship overlaps any of the enemies and ends the game if it does.

```
move direction = save (collideShip (moveObj direction ship))
```

It also accepts a *restart* update, which sets the game state back to that of an initial game, and a *quit* update which sets the GameOver Boolean value to True.

```
quit = save (lasers,enemies,ship,score,True)
restart = save initgame
```

We are able to decouple move updates, from time based changes here, something that is impossible in Haggis.

3.12.2.2. The Screen Component

The *Screen* component creates the whole screen appearance. It is significant here that though the *GameState* RH is more closely associated with the *Screen* component it must reside in the root of tree. This occurs because components can only request values from parents in the tree. It is therefore a feature of Clock that RHs may appear at initially confusing positions in the tree hierarchy.

The *Screen* component accepts keypress inputs and tick updates. The former moves the shape in a given direction; the latter sends a timer update. These only happen when the game is not halted.

```
gamehalted = paused || gameover
keypress k = if gamehalted then
               noUpdate
             else if k == 'q' then
               move Left
           else ...

tick = if gamehalted then noUpdate else timer
```

For its view it uses an instance of the *ShipView* subview, and one *EnemyView* and *LaserView* for each currently active object. Dynamic creation of objects is therefore done implicitly, by defining a mapping of the current view.

```
view = if gameover then Views [largeText "GameOver",game]
       else game

game = Views (shipView "ship":
                (map enemyView enemies ++ map laserView lasers))
```

3.12.2.3. Discussion

The Clock approach allows for a fairly declarative implementation. Components each have views and these can be easily composed, using a set of declarative combinators.

Clock lacks any explicit temporal operators except for the simple timer update. Within any one component, as each event is evaluated, all values in the tree are constant. Sequencing occurs, to some extent, by passing updates up the tree, firing constraints back down the tree. It is significant that the pause behavior had to be programmed directly into the shape component. This means that the architecture is not entirely modifiable. Support for a disable operator, to disable a subtree would have been useful here.

3.12.3. QOC Editor and file manager

The third Clock example is the QOC editor. The Clock architecture for this example is shown in Figure 16. Again the architecture structure is based on the visual appearance of the system. Each user has an editor view (*Editor*) which consists of the workspace and the tool bar (which has the save, create and view buttons.) The workspace consists of QOC windows. These each consists of a set of nodes (Questions, Options and Criteria) along with connecting edges and possibly a rubber band line (if two nodes are being connected.) A node has a label, a local editor (*Annotations*) for the user and a view of the shared log for that object (*SavedAnnotation*). An edge will also have a log, a view and possibly a label. The system is built from a set of relations. Some data is local to individual users, some is global to all users. For instance, at the root level are RHs that hold information about all the nodes, edges, positions and the shared log. The list of visible QOCs can be filtered. At the QOC window level the list of nodes will be filtered, to consist of those that are visible because of the current view.

We partition the state into RHs. All information about the current QOC set is held in the *QOCData* RH. This has operations to add and delete QOCs, and to add and delete nodes and edges from a given QOC. This single RH is very monolithic. Unfortunately, we cannot define RHs that contain other RHs. If we could we might prefer to consider the QOCData RH to consist of a lists of QOCInfo RHs, that maintained the state of each individual QOC.

The list of visible QOCs and nodes and edges will be filtered. As an example, we show how to display the list of visible QOC windows.

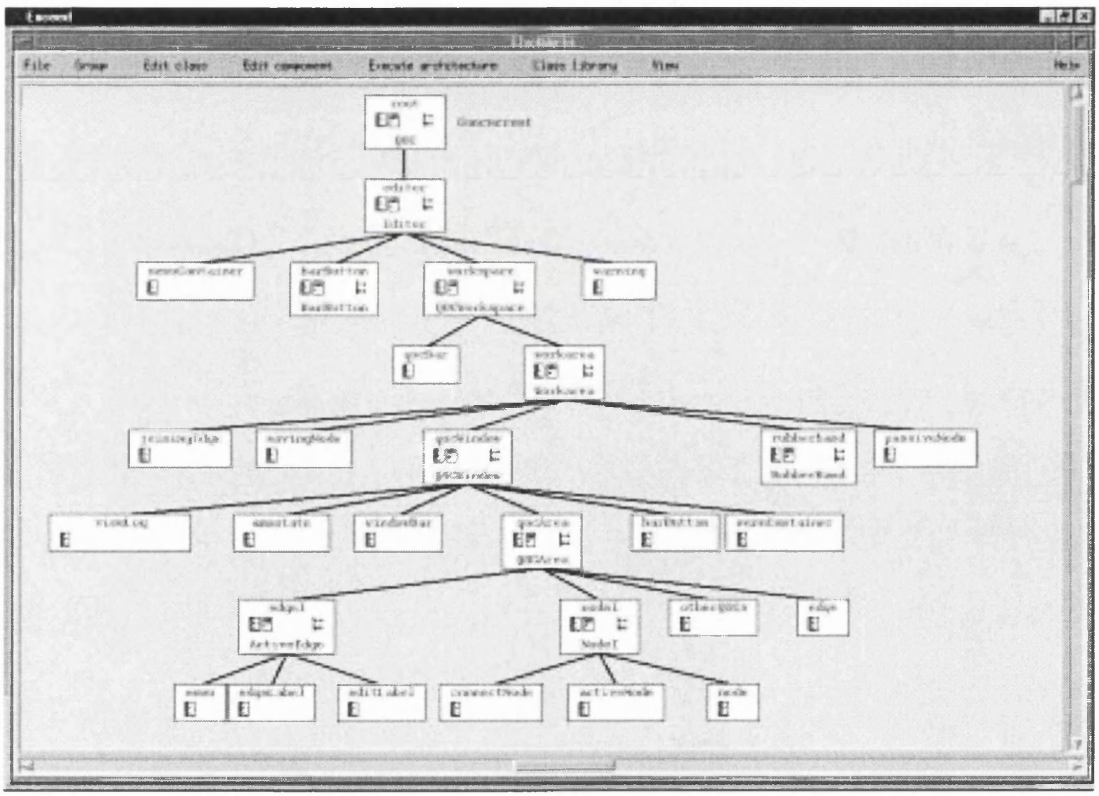

Figure 16 - The Clock Architecture for the QOC Editor

We define an RH to contain the list of invisible window names. This has updates to add and delete a QOC name. It also provides a request that defines whether a given QOC is visible. We then define the workspace view as follows.

```
view = Views (map windowView (filter isVisible allqocs)
```

The use of constraints here is very powerful. The dynamic creation and deletion of objects is handled automatically based on the current state of the underlying application.

When a QOC is deleted by a user, we can remove it from the invisible list. However, if it were to be deleted by another user there would be no mechanism to remove it from this list. Fortunately this does not make the program incorrect, as subviews and therefore QOCs must have persistent unique String names. Unfortunately, it does make the visible list redundantly long which makes it less efficient. This problems results from the fact that we cannot receive events from higher in the tree, nor can we make a RH's data dependent on that of another RH.

We can define the behavior of a moveable component as follows. It receives mouse-button and motion input. It can be moved and when moving is highlighted.

```
mousebutton Down = startmoving myid
mousebutton Up = stopmoving myid
motion (x,y) = setposition myid (x,y)

view = if ismoving myid then
          draw highlight icon
       else
          draw unhighlighted icon
```

The ability to use constraints here to describe the view is powerful. It makes it easy to see the relationship between the state and view.

3.12.4. Discussion

The ability to use constraints to define an interface view is a very powerful feature of Clock. It also provides good support for application, interface separation.

Clock does however have some problems. The first set of these, result simply from its implementation as a research prototype.

1. The graphics that can be produced by Clock are fairly limited.
2. There is no way to access code written in other languages from Clock. All application code must therefore be developed in Clock. This severely limits its applicability to other systems.
3. The underlying functional language does not include type checking, and is in general fairly poor. This means that defining application objects can be cumbersome and may be slow. The optimisation work has instead gone into the network support and interactive graphical support.
4. Clock has no module structure, making it difficult to define modular code that can be reused in different request and event handlers.

The design of Clock also has some more fundamental problems.

1. Clock does not support behavioural values that change as functions of time, except through the *tick update* mechanism. This reduces the expressiveness of the language. We are also unable to define mutually recursive behaviors. This makes it difficult to define a mutually recursive behavior modelling the trajectory of a moving object, where the speed, location and acceleration might be mutually dependent.
2. Clock relies on constraints and updates to implement all necessary temporal operators. This may make it difficult to understand complex temporal relationships. This makes it more difficult to define real-time functions. For instance, in our Air Traffic Control case study (introduced in Chapter 2), we need to be able to define time out predicates that result when a message has not been received by an aircraft
3. The inability to define composite Request Handlers, which consist of other Request Handlers, makes structuring complex application behavior difficult. In particular, where there is a complex application that is used by all components, we end up with one very large, monolithic request handler.
4. The use of untargeted updates and requests, can be useful. However, sometimes this can also be very cumbersome. For instance, it makes it impossible to have two instances of a request handler in scope at the same time.

5. The use of unique String names to distinguish instances of components is fairly clumsy, and makes mistakes more probable. It also makes it difficult to finally garbage collect a component, as there is no clear way to tell if its name (and it) will ever be used again. This makes the implementation inefficient.
6. A Request Handler can neither hear update events from higher in a Clock tree, nor can it depend on a Request Handler defined higher in a tree. This makes the definition of components, such as the view filter component, quite awkward.
7. In general, the Clock tree structure, though useful can be overly cumbersome. In particular, it would make it impossible to define a recursive, fractal view, where a component class could contain instances, of its own class. For instance, we might want windows that could contain windows. This would require a more general graph structure rather than a simple tree structure.
8. All events in Clock travel up the tree. It is therefore impossible for parent components to filter input, or disable children. For instance, to produce a window that disabled all its child components on the press of a button would require the redesign of all child components to include a disable operator. The possibility of instead taking a snapshot of the child's view and using this until the window were re-enabled might solve this problem.

3.13. Functional Reactive Programming

Fran (Functional Reactive Animation) [44] is a language for constructing interactive animations. It uses a high-level modelling approach that allows programmers to describe what an animation should look like, not how it should be implemented. Fran introduced the Functional Reactive Programming (FRP) approach. The key notions that FRP introduces are *behaviors* and *events*. *Behaviors* are time-varying, reactive values, while *events* are streams of values that occur over time. The FRP approach has also been applied to a number of other application areas including robotics programming [155]. Courtney has begun to apply the FRP approach to Java, to simplify the creation of Java Beans based interactive systems [33]. His intention is to provide a tool to visually connect behaviors and events and link them to interactive components. This work, though very interesting, is still in its infancy. The range of combinators is very limited and is restricted to systems with static numbers of components.

3.13.1. Fran benefits

There are five key aspects that makes it a good candidate for forming the basis of a User Interface development language.

1. **Behavioral Modelling.** Fran uses first-class behavior values to model changing values in an animation. A behavior value is a value that changes over time. It can be thought of as

   ```
   type Behavior a = Time -> a
   ```

   As an example we can make a circle that follows the wave path shown in Figure 17. Its position is a function of time; it moves along the screen as time passes, and up and down as the *sin* of time.

   ```
   moveXY time (sin time) circle

   moveXY :: Behavior Double
          -> Behavior Double
          -> ImageB -> ImageB

   time :: Behavior Double
   ```

   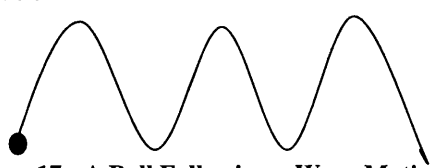

   Figure 17 - A Ball Following a Wave Motion

2. **Event Modelling.** Events, like behaviors, are first-class values. An event is a stream of values that occurs at discrete points in time. It can be thought of as

   ```
   type Event a = [(Time,a)]
   ```

We can use events to model happening in the real world, such as button presses; and predicates based on behavior values, such as collisions between objects. For instance, a left button press is simply `lbp u`, where u is a User argument. An event that occurs once, when the time is greater than 5, is `onceE (predicate (time >* 5) u)`. They can be combined as `lbp u .|. onceE (predicate (time >* 5) u)`. The type signatures for the functions we have used are shown below.

```
lbp ::User -> Event ()
predicate :: Behavior Bool -> User -> Event ()
onceE :: Event a -> Event a
(.|.) :: Event a -> Event a -> Event a
(>*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
```

3. **Declarative Reactivity.** Much of the power of Fran comes from the interaction of behaviors and events. We can define "reactive behaviors", that change as events occur. We can therefore give a declarative rather than an imperative semantics to state. For instance, we can describe a colour-valued behavior that starts out white, and then changes to red when the left button is pressed, and changes to green when the right button is pressed.

```
white `stepper` (lbp u -=> red .|. rbp u -=> green)

stepper :: a -> Event a -> Behavior a
```

4. **Declarative Composition.** Animations can be constructed compositionally. We can create two balls following wave motions, one that moves as the *sin* and one as the *cos* of time.

```
moveXY time (sin time) circle
`over`
moveXY time (cos time) circle
```

Note that these two animations evolve concurrently, and yet are described in a simple, deterministic manner.

5. **Models and Views.** We can define a behavior to represent the state of an animation and then define a function that transforms this state into an image. For instance, we can describe a moving object abstractly in terms of a data type with behaviors[4]. It has a colour and a location.

```
data Object = Object (Behavior Point2) (Behavior Color)
```

We can then define a function that turns this colour and location into an image.

```
view (Object pos col) = move pos (withColor col circle)

move :: Behavior Point2 -> ImageB -> ImageB
withColor :: Behavior Color -> ImageB -> ImageB
```

The features discussed above provide a powerful approach to building interactive systems. We could describe the state of an interactive system as a behavior, reacting to user input events. These components could be easily composed in a declarative manner. We could also support a Model-View style of programming where the state of the application is described abstractly and the appearance described as a function of that state.

3.13.2. A Case study with Fran

Fran is an animation environment, rather than a more general user interface toolkit. This makes it difficult to define the QOC editor in Fran. We can, however, implement the space fighter game in Fran. I will now discuss the most important aspects of this implementation.

---

[4] In Haskell new algebraic data types are defined using the keyword `data`. A datatype has a name and one or more constructors. Each constructor may have one or more values associated with it. For instance, the definition `data Alt = AltA String | AltB Int` , defines a data type named `Alt`, with two possible constructors `AltA` and `AltB`, each with a single value associated with them.

We define the Game data type to represent the state of the game. This consists of a ship, an event generating new lasers, an event generating new sets of enemies and an event generating an occurrence when the game ends.

```
data Game = Game Ship (Event Laser) (Event [Enemy]) (Event ())
```

We can therefore have one simple abstract model of the state of the application.

We use the Renderable class to define how the Game state is displayed. We display the ship, above the lasers, above the enemies. This display continues until the game ends, after which we show the text "Game over".

```
instance Renderable Game where
 render (Game s ls es end) =
    render s `over` render es `over` render ls
    `untilB` end -=>
    (stretch 5 $ withColor white $ showIm "Game Over")
```

Note that when applied to an Event, render applies the render function to every new occurrence of an object and overlays it above all the previous instances. We therefore have one set of functions that define the view in terms of the application.

We define the game itself as follows. It takes the user input, an event generating move occurrences, and an event generating fire occurrences. We create a ship. The game ends when the ship crashes. When a fire event occurs, we snapshot the ship location and current time, and generate a new laser. Every `enemyTime` seconds, we generate a new set of enemies. Lasers and the ships die when they collide with enemies; enemies die when they collide with ships. We handle this by generating an abstract collision set, or `ColliderB`.

```
game :: User -> Event (S.Point2 -> S.Point2) -> Event () -> Game
game u moveE fireE = Game shipD mkLaser mkEnemy (crashE shipD)
  where
   shipD :: Ship
   shipD = ship u moveE hitEnemies

   mkLaser :: Event Laser
   mkLaser = afterE (withTimeE_ fireE  `snapshot` (loc shipD)) u
             ==> \((t,p),u) -> laser u t hitEnemies p

   mkEnemy :: Event [Enemy]
   mkEnemy = afterE (withTimeE_ (alarmE u (startTime u) enemyTime)) u
             ==> \(t,u) -> map (enemy u t hitLasers ) enemyBatch

   hitEnemies :: ColliderB
   hitEnemies = eventCB (mkEnemy ==> mergeCBs . map hitObj)

   hitLasers :: ColliderB
   hitLasers = eventCB (mkLaser ==> hitObj)
```

The ability to generate events that go off at a particular time interval, and the ability to make objects depend on time is important. It allows us to specify the behavior of the game at a very high level of abstraction. The ability to define a collision behavior as an abstract value with a simple set of combinators is also useful. It makes the collision definition more compositional than would be possible with an imperative definition.

We define lasers, enemies and the ship in terms of a more general Object type. These have a location, bounding box, a Boolean behavior defining whether they are dead, and an event that generates an occurrence when they crash. Again we therefore have a simple abstract model of an object.

```
data Object = Object {locO :: Point2B,bboxO :: RectB,
                      deadO :: BoolB,crashOE :: Event ()}
```

We create an object as follows. It has a bounding box, based on the current location and size. It is dead once a crash event has occurred. It crashes when its bounding box collides with the relevant collision set. Again this gives us a very simple definition of the behavior of an object within the system.

```
mkObject :: User -> Point2B -> SizeB -> ColliderB -> BoolB -> Object
mkObject u loc size collide die = Object loc bbox dead crashE
 where
   bbox   = rect2B loc size
   dead   = falseB `untilB` crashE -=> trueB
   crash  = (testCB collide $* bbox)
   crashE = onceE (predicate u crash)
```

All objects are members of the Obj class. We can display any member of this Obj class in the same way. We display its image, at its current location, until it crashes, after which we display an empty image.

```
instance Obj a => Renderable a where
 render a = moveTo (loc a) (image a) `untilB` crashE a -=> emptyImage
```

This implementation is declarative and very modular, consisting as it does of a collection of dynamic components. It also has separation between the application Game state, and the view, which is defined, using `render`, as a function of this state.

3.13.3. Fran problems

Fran as it exists in its basic form has two serious conceptual restrictions that must be overcome to extend it to interactive systems design.

1. **Hierarchical input.** The most significant problem with Fran is that it does not provide any support for constructing hierarchical interactive displays. All user input is accessed through the *User* argument passed to the animation. This represents input at the level of the entire graphics window. There is no way to access interaction from only a single component, such as an individual button on the screen. The only conceivable mechanism would be to use global mouse coordinates and calculate whether they were within the bounding box of a given object. This approach does not lend itself at all to building hierarchical collections of components, each with their own coordinate systems. This is not usually a major problem in animations, however, the notion of individual interaction objects is critical to the development of standard user interfaces.

2. **Dynamic collections.** Fran provides *Behaviors*, that is values that change over time. We could therefore imagine having behavior collections of objects. For instance, we could display a dynamic list of objects. However if we were to render such a behavior collection, each time an element were to be added *the entire collection would have to be redrawn.*. This would be prohibitively expensive if we needed to continually recreate complex compound collections. In our space fighter game, we defined the lasers and enemies as *Events* rather than *Behaviors*, to make them useably efficient. This prevents us from being able to snapshot the current state of the lasers or enemies. Fran requires some notion of an incremental behavioral collection that could be both viewed as a behavior and efficiently and incrementally rendered.

3.14. Pidgets

Pidgets [178] is a system that has been designed along similar lines to FRAN. It uses an approach called a "monad of imperative streams" to provide a temporal language. It supports the creation of a group of dynamically evolving, concurrent objects. The basic concept in Pidgets is a Stream. A Stream produces values, like a Fran event, and has a current value, like a Fran behavior. The current value of a stream is the last one it produced. Unlike Fran behaviors, a stream will therefore not have a current value, until it has produced its first value. Streams may also perform IO actions, and so carry out some imperative command. A program is evaluated at a series of steps, in a synchronous manner. At each step a stream may produce a value, an stream therefore produces a list of values over time. Components do not compete for users events. Instead each event is passed to every object. For instance, every time the mouse position changes, each object is informed of its new position.

We can implement the colour valued behaviour, from the Fran example (Section 3.13.1), as follows:

```
colourStep :: St Colour
colourStep = until (next mouseButton)
                   red
                   (until (next mouseButton)
                          green
                          colourStep)
```

A value of type `St a`, is a program that creates a *Stream*, that produces values of type `a`. The `until` operator is similar to Fran's. The `next` function guarantees that we must wait for the next step when the `mouseButton` is pressed. We can also have Boolean valued predicates, for instance, we could restrict `colourStep` to only produce values until 10 seconds had passed:

```
restrictedColour :: St Colour
restrictedColour = until (lift1 (> 10) time) colourStep nil
```

Here we say that `until` the current `time` is greater than 10 seconds, produce values from `colourStep`. After that we produce `nil`, an empty stream.

A stream can be considered in terms of its status, that is its current value. Alternatively, a stream can be considered in terms of events: we can also check to see when an object produces a new value. For instance, we could write a function that writes to a file every time the colour (from `restrictedColour`) changed.

```
fileWriter :: String -> St ()
fileWriter file = do
    c <- restrictedColour
    writeFileIO file (show c)
```

In this case, the syntax `c <- colourStep` means for every value produced by `restrictedColour` perform the following action. When `restrictedColour` produces the value `nil`, `fileWriter` will cease.

Imagine that we wished to define instead a stream that on every keypress, sampled and displayed the value of `colourStep`. We might try to define this as follows.

```
keyColour :: St Colour
keyColour = do
 k <- keyboard
 current colourStep

current :: St a -> St a
```

Every time a key is pressed, we sample the `colourStep` stream using `current`. Unfortunately, every time a key is pressed, the `colourStep` stream is restarted, so `keyColour` will only ever produce the value `red`. To correct this we must use `start`.

```
keyColour :: St Colour
keyColour = do
 c <- start colourStep
 k <- keyboard
 current c

start :: St a -> St (St a)
```

It decouples the steps in which a stream is started and accessed. It spawns a substream running concurrently, and returns a program that may be used to create sink streams for the started stream.

Pidgets includes a Widget library. We can define our example "Counter" as follows.

```
counter :: St ()
counter = do
 w <- newVar
 runWidget "counter" $
  let lbl = textSt (lift1 show (var v))
      btn1 = pushButton (updNext v increment) (text "inc")
      btn2 = pushButton (updNext v decrement) (text "dec")
  in above (lbl (beside btn1 btn2))
```

We create a stream variable that stores the state of the counter. This has an action to update it, and a function, `var`, that accesses its values. Instead of accessing its single value, we access a stream of values, representing its state for all time.

```
newVar :: St (Var a)
updNext :: Var a -> (a -> a) -> St ()
var :: Var a -> St a
```

We define our interface in terms of the Widget type. We run a Widget in a window, with a given title, using `runWidget`. We create a text label using `textSt` and text. We create a button using `pushButton`. This takes an action to perform on every click and an appearance widget, and produces a button widget. By passing in the input action, we are able to define a generic Widget value, and so easily compose them. As with Gadgets, this allows for good separation between "semantic wiring" and geometric composition.

```
runWidget :: String -> Widget -> St ()
pushButton :: St () -> Widget -> Widget
textSt :: St String -> Widget
text :: String -> Widget
above,beside :: Widget -> Widget -> Widget
```

The Pidgets approach therefore has a number of positive features. It provides many of the same advantages as FRAN support for: constraints, powerful temporal operators, composition of dynamically evolving images, and status and event considerations. The Widget library provides some useful concepts, such as a generic untyped Widget value.

However, it has a number of significant problems. Pidgets unifies three very separate concepts, producing a value, having a value, and performing an action. This makes the type signatures appear initially very confusing on many of the above functions. We cannot tell from the type signature whether a value of type `St ()` is a single action, or whether it produces a stream of `()` values. We cannot use the type system to check that we are treating these different concepts correctly. There is also no difference in type between an unstarted and a running stream. The `start` and `next` functions can be conceptually very confusing. In particular, when a stream also performs an `IO` action, we must be careful with any use of start. By combining these concepts the choice of operator name can be confusing. If we're treating a stream as having a value, then the name `current`, makes good sense. It returns the current value of the stream. However, if we're treating a stream as producing values, then we use `current` to get only the first produced value from a stream. This separation is also obviously altered depending on whether the stream has been started earlier. If the stream has been started earlier it will almost certainly have already produced a value, and current will return that value. If it is instead started at the point of use, then it may not produce a value for several steps. In this case current, will not return a value immediately.

Pidgets also provides no support for dynamic displays with changing numbers of components. The Widget library only really provides support for creating new windows, by performing successive `runWidget` actions. This seriously restricts its potential.

3.15. Requirements for Declarative GUI Languages

Based on my analyses, presented in previous sections, there are a number of important requirements for user interface programming languages. These do not represent a completely comprehensive set of requirements and in particular do not concentrate on high level notions about the usability of the language. Instead they represent a set of design goals derived from carrying out numerous small examples and the large case studies with Haggis, Clock and Fran.

3.15.1. High level and Declarative

In order to implement high level specifications, and allow designers to perform high level prototyping, languages must themselves be very high level. High level functional concepts should be supported such as higher order functions, and polymorphism. Languages should also support the concept of behavioural values that may change over time, with predicates on these values. This enables both status and event phenomena to be dealt with. Finally, languages should support the dynamic creation and deletion of objects in a safe, referentially transparent way. In particular, throughout the language the programmer should be able to use a declarative style, specifying what an interactive system should be like not how this should be achieved.

3.15.2. Declarative Concurrency

Languages need to support concurrency, but at a high level of abstraction. Approaches where programmers must explicitly perform message passing between components, and handle the intricacies of concurrent programming have problems. Instead, languages should support dynamically evolving objects, where communication can happen via constraints. Concurrent support for multi-user systems is necessary. However, there should be as little reliance on explicit concurrency as possible. This should allow errors to be visible at compile time.

3.15.3. Compositional

Systems should support a compositional style of programming. In particular, it should be possible to form complex interactive components out of primitive ones and use them in an equivalent way. To aid this, languages should support easy composition of objects, each with local state.

Graphics need to be specified in a declarative way. There should be support for a wide variety of possible appearances. Composition of dynamically evolving pictures should be possible, as is permitted by Pidgets, Fran and Clock.

3.15.4. Component based application/interface separation

Languages should support the structuring of a system into components. If a conceptual architecture is used as the basis then we can guarantee application/interface separation. This sort of support is provided by Clock, with its basis in the MVC model. To allow this there should be good support for separation between "semantic wiring" and geometric composition.

3.15.5. Visual Languages & Tool support

Where helpful, languages should aid development, in a visual way. Designers should be able to consider architectures graphically. This should be possible through direct manipulation. They should also, if possible, be able to design interfaces visually. This allows designers to easily design attractive interfaces.

3.15.6. Scalability

The language should scale to easily handle large examples. UI languages often appear very compositional, until they are used to structure very significant systems. In particular, it should not be necessary to radically abandon the general programming paradigm, to handle big examples.

3.15.7. Efficiency

While languages should be easy to use, they also have to produce efficient systems. Otherwise, while systems may be easy to develop, they will be of little use. This relates to the scalability issue. It should be possible to run large programs without waiting noticeable periods of time for screen updates to occur.

3.15.8. Platform Independence

A prototyping language should support the development of platform independent implementations. This can be aided by building on a platform independent underlying toolkit, such as Tcl-Tk instead of building from the ground up as has been done with most previous functional GUI systems.

Chapter 4 - FranTk: A New Approach

4.1. Introduction

The programming language, FranTk, presented here represents a new approach to User Interface programming. The core FranTk library is the most important element of the prototyping environment discussed in this thesis. There are two additional elements that make up the approach shown in Figure 18.

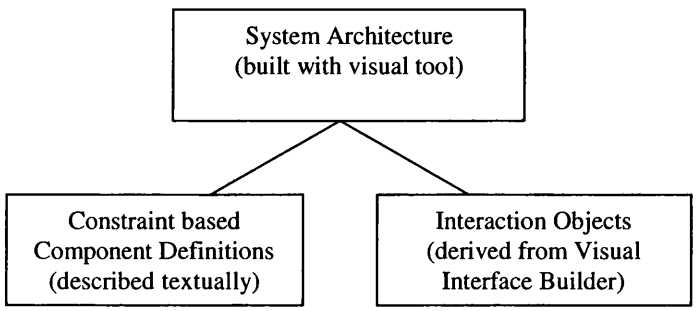

Figure 18 - Elements of the Thesis Prototyping System

The prototyping environment supports a visual interface builder, and system architecture manager. The static components of an interface can be constructed visually with an interface construction tool. A structured architecture may also be developed. This architecture can be built visually. This supports the designer's understanding of the system structure. This architecture breaks the application down into a tree of interaction components, and is based on that used in Clock. Components are described textually in terms of FranTk combinators. Code can be automatically generated for each static widget that has been built with the visual interface builder, to allow the widgets to be linked in with FranTk code.

FranTk makes use of a binding to the popular Tcl-Tk language [147], to provide a widget library. Tcl-Tk provides a powerful set of platform independent widgets. Prototypes can therefore run on UNIX or the Windows family (95/98/NT/2000). To enable multi-user interaction, it makes use of (and needs) the X Windows client-server architecture to allow several client interfaces to be run on different machines from one UNIX server.

The core FranTk language has been released as a publicly available software library. It is therefore intended for use by other people. However, the visual tools were only developed as proof-of-concept prototypes and are not in a state to be used by others.

The remainder of this Chapter introduces the core FranTk language. This chapter uses a number of small examples[5]. These examples serve a different purpose from the larger case studies discussed in Chapter 2. They introduce important features of FranTk. Chapter 6 will discuss how FranTk handled the Thesis case studies themselves.

4.2. FranTk contributions

The major new contributions in FranTk are as follows:

- FranTk lifts Fran's behaviors and events to widgets. This is the key to the declarative style of programming. The appearance of a dynamic widget can be defined once *"for all time"* in terms of FranTk combinators.

[5] This chapter assumes some familiarity with Haskell. Some elements of the basic syntax were introduced in Section 3.7. Some attempt has also been made to introduce further syntax used in this Chapter. Readers requiring further assistance are again directed to [199] or [94].

This chapter also makes use of a number of standard Functional Reactive Programming combinators. A full summary of these combinators is given in Appedix A.

- FranTk extends Fran with support for hierarchical interactive displays, allowing access to input from individual components rather from one monolithic window.
- FranTk separates visual composition from "semantic wiring". These two concepts are fundamental to GUI programming. The first involves geometric composition. For instance, placing one widget above another. The second involves connecting user input from a widget to the application code. This separation is made possible by the introduction of *listeners*, consumers that respond to user input. FranTk provides an algebra to compose these listeners in a functional style.
- FranTk provides good support for dynamic and static interfaces. This contrasts with many of the systems discussed in the previous Chapter where it can be difficult to construct systems with a dynamically changing number of components as they frequently require a very imperative and sometimes cumbersome style of programming.

4.3. The Basic Concepts

We begin with a simple example to introduce some basic concepts. Figure 19 shows two labels, composed above each other. The top label shows the text "Hello World", the bottom label shows the time in seconds since the program was started. The concepts necessary to produce this interface will be presented in this section.

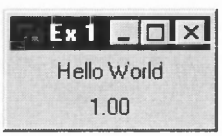

Figure 19 - A Simple Example in FranTk

4.3.1. Introducing Components

The basic conceptual notion in FranTk for handling interaction objects is the *Component*. For instance, in our example we have two label components displayed one above the other. They appear in a window component.

A Component is an action that produces a `WidgetB`.

```
type Component = GUI WidgetB
```

This definition uses the GUI monad, which is an extension of the standard IO monad. Values of type `GUI a` represent actions that may have some side effect on the user interface, such as creating a label, and return a value of type `a`.

A `WidgetB` is an abstract data type representing primitive Tcl toolkit widgets. A `WidgetB` may in fact be made up of several primitive tcl widgets, and may be dynamic, changing its appearance over time. *A* `WidgetB` *is therefore conceptually a widget behavior.* However, as with Fran's `ImageB` type (image behaviors), `WidgetB` is an abstract data type to allow an efficient implementation.

As well as basic Components such as labels and buttons, there are top-level window components that contain basic components. We therefore have the concept of a *WComponent;* a Window component.

```
type WComponent = GUI WindowWidgetB
```

This is an action that produces a `WindowWidgetB`, which is an abstract representation of a window behavior (that will contain widgets).

4.3.2. Configuring Components

In our example we have two labels with different appearances. One has a static appearance; one a dynamic appearance. We create a label using the `mkLabel` function and configure it using the `text` and `textB` functions.

```
mkLabel :: [Conf Label] -> Component
text :: Has_text w => String -> Conf w
```

We use `mkLabel` to make a label. It takes a list of configuration information, in this case, some text to display. As with TkGofer (Section 3.8.1) we use type classes to guarantee that only the correct configuration information can be applied to any widget. The `text` function takes a String and returns a configuration option that can be applied to any object that is a member of the `Has_text` class. This class includes labels, as they are capable of displaying text.

We therefore define the static "Hello World" label as follows.

```
label1 :: Component
label1 = mkLabel [text "Hello World"]
```

In FranTk, we extend basic configuration with dynamic configuration options. Instead of taking a static value, a widget can be given a dynamic behavior value. This approach is the key to the declarative nature of FranTk. Rather than having to carry out imperative updates to change a component's appearance, we can define, using a behavior, what it will look like *for all time*.

We can therefore define the timer label as follows.

```
label2 :: Component
label2 = do
 time <- timeTick 1
 mkLabel [textB (lift1 show time)]

textB :: Has_text w => Behavior String -> Conf w
timeTick :: Time -> GUI (Behavior Time)
lift1 :: (a -> b) -> Behavior a -> Behavior b
```

The function `timeTick` creates a behavior that represents the time, and changes at a given frequency. In our example, the time value changes every second. The function `lift1` maps a function over a behavior, to yield a new behavior. This shows the benefit of the GUI representation of a `Component`. We can create some local state for a component while still thinking of it as a value.

4.3.3. Composing Components

We now have the definition of the two labels. We can compose them together to generate a new component using `above`.

```
newlabel :: Component
newlabel = label1 `above` label2

class Packable w where
 above :: w -> w -> w

instance Packable Component
```

Though components represent imperative actions that will each produce a widget, we can treat them as an abstract value and so compose them declaratively. This satisfies our aim of supporting a compositional style of programming in FranTk.

As with Pidgets (Section 3.14) and Gadgets (Section 3.9.2), the representation of widgets as untyped components makes them easy to compose. In contrast, systems which use typed widgets are less easily composable. For instance, in Haggis or TkGofer a button and label are of different types. This typing was required to allow access to user input from the component, and to apply changes to the component, such as resetting the label, later on. This approach makes it more difficult to geometrically compose a list of widgets as they must be transformed to an untyped display object first. In FranTk, all the information necessary to define a component is passed in as a set of parameters so this sort of extra

transformation can be avoided. This design choice is very significant and is discussed further in Section 4.15.

We can place a component in a window using the `mkWindow` function.

```
win :: WComponent
win = mkWindow [title "Ex1"] newlabel

mkWindow :: [Conf Window] -> Component -> WComponent
title :: String -> Conf Window
```

Again this takes a list of configuration information and the component to display, and produces a window for that component.

4.3.4. Configuring composite components

As well as applying configuration options to individual objects, we can apply them to composite widgets. For instance, we might wish to make the background colour of our whole window blue. It would clearly be unfortunate if we needed to set each individual object's background colour, as this would make our programming style significantly less compositional. We can change the background colour of our example to blue as shown below.

```
win2 :: WComponent
win2 = withStyle [background blue] win

class HasStyle w where
  withStyle :: [Conf Style] -> w -> w
```

This uses the `withStyle` function, which is a member of the `HasStyle` class. All FranTk component types, including `Component` and `WComponent`, are members of this class. The `withStyle` function takes a list of `Style` configuration options. These consist of all options that it makes sense to apply to a composite widget, such as formatting options; but not individual options such as the text configuration option. These style options include a disable option, which disables the entire composite widget. They are applied to every widget in the composite component, which can handle that sort of configuration option. For instance, we can set the font size for a composite widget. Clearly only those objects that can display text will need to have this option applied to them. If options are defined at several levels, then the one at the lowest level in the tree will be applied.

4.3.5. Rendering Components

Now we need to render this window component onto the screen. We do this using `render`.

```
render :: WComponent -> GUI ()
```

Finally, to run the GUI actions that we have produced we use start. This runs the action and then starts up the Tcl-Tk event loop. This event loop will run until the graphical user interface quits, at which point it will return.

```
start :: GUI () -> IO ()
```

As `start` and `render` are often used together there is a composite function display.

```
display :: WComponent -> IO ()
display = start . render
```

We can therefore define main as follows.

```
main :: IO ()
main = display win
```

4.4. Interactive Components – Representing State

So far we've dealt with an interface with a changing appearance but with no interaction. This section demonstrates how interaction is handled in FranTk.

Consider the interface shown in Figure 20. It shows the simple "Counter" with a label, an increment and decrement button, and a slider (known as a scale widget). This widget has a current value shown by the slider and the label. Pressing the increment or decrement button, or moving the slider will change this value. We therefore have multiple views of the same data.

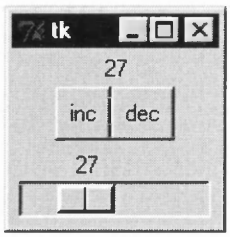

Figure 20 - An Interactive Example in FranTk

4.4.1. Representing State with BVars

To represent the state in the example we use a `BVar`. A value of type `BVar a` represents some abstract mutable state of type a. It can be thought of as a *Behavior Variable*: that is a variable that contains a behavior.

```
data BVar a
```

We can create a new `BVar` within the `GUI` or the `IO` monad. Most commonly we use them within the `GUI` monad.

```
newBVar :: a -> IO (BVar a)
mkBVar :: a -> GUI (BVar a)
```

It is possible to get a behavior from a `BVar`.

```
bvarBehavior :: BVar a -> Behavior a
```

The behavior therefore represents the value of the counter at any give point in time. It is possible to get an event from a `BVar`

```
bvarEvent :: BVar a -> Event a
```

The event from a `BVar` generates an occurrence every time the value of the `BVar` is updated. In our example we would therefore represent the state of the counter as a value of type `BVar Int`.

The use of BVars here is similar to the use of stream variables in Pidgets (Section 3.14). They are a fundamental and important feature of FranTk. They provide us with a mechanism to represent state, but to use it in a functional style.

4.4.2. Using State – With Behaviors

What can we do with a behavior? We can tell the label and slider to display the behavior values that we get from the `BVar`.

```
lbl :: BVar Int -> Component
lbl m = mkLabel [textB (lift1 show (bvarBehavior m))]
```

As shown in section 4.3.2 we can tell a label to show a string behavior using `textB`, and we transform the integer behavior into a string behavior using `lift1`.

We can tell the slider to use the value of the BVar with scaleValB. This sets the value of the slider to that of an integer behavior. (We will fill in the rest of the definition later.)

```
scale :: BVar Int -> Component
scale m = mkHScale [scaleValB (bvarBehavior m)] (..)

scaleValB :: Behavior Int -> Conf w
```

We can therefore easily provide multiple views of the state of an application.

4.4.3. Updating State- With Listeners

We can set the value of a BVar using its Listener.

```
bvarInput :: BVar a -> Listener a
bvarUpdInput :: BVar a -> Listener (a -> a)
```

A listener is an abstract type but it can be thought of as `Listener a = a -> GUI ()`. A value of type `Listener a`, is a function, that given a value of type a, performs a side-effecting GUI action with it. Listeners are therefore consumers of values.

The listener accessed by bvarInput updates the BVar with its given value. This will alter the value of the BVar's behavior and generate an event occurrence. The listener accessed by bvarUpdInput updates the BVar by applying the given function to its current value.

We can therefore complete the definition of the slider as follows.

```
scale m = mkHScale [..] (bvarInput m)
mkHScale :: [Conf Scale] -> Listener Int -> Component
```

The function mkHScale takes a listener argument, which is passed the current value every time the slider updates the BVar with its changed value. The slider simply updates the value of the BVar with its changed value.

The introduction of listeners is a very important design choice. Initially this choice may seem strange. The use of behaviors and events encourages a more functional style of programming. However, the use of listeners introduces an imperative concept into this functional approach.

The introduction of listeners brings an important benefit. In a similar manner to Pidgets, we give component-making functions a consumer (listener) argument which allows them just to yield their visual aspect in the form of a Component. This approach makes geometric composition simpler. The alternative would be to return a pair of visual and semantic handles, in the form of a Widget and an Event providing access to all user input on that widget. This alternative makes component composition more complex. To compose two components we would now require to compose their Widget and Event parts. This style can become tiresome when composing complex collections of components, because programmers are forced to continually compose and break down compound events. We will return to this important design choice in Section 4.15.

At this point it is perhaps also useful to compare a FranTk BVar to a Java Beans *Bound Property*. A Bound Property has a set method which updates its state; this role is fulfilled by the BVar's Listener. It is possible to get the value of a *Bound Property*; this is achieved through the BVar's Behavior which can be sampled at any given time. Finally, it is possible to add a PropertyChangeListener to a Bound Property to hear about changes. This role is fulfilled by the BVar's Event. A FranTk Event is perhaps better understood as an *Event Source*. It is possible to add Listeners to it to hear about changes. It is important to note that though BVars and JavaBean Bound Properties have similarities, as we shall see, there is an algebra of combinators for Listeners, for Behaviors and for Events which makes working with them much more succinct than with their Java counterparts.

4.4.4. Composing Listeners

FranTk introduces combinators that allow listeners to be composed in a functional style. As an example, consider the definition of the increment button in our example.

```
incb :: BVar Int -> Component
incb m = mkButton [text "inc"]
            (tellL inc (bvarUpdInput m))

inc :: Int -> Int
mkButton :: [Conf Button] -> Listener () -> Component
```

The function `mkButton` takes a list of configuration information for a button. This defines its appearance. Note that Button is also an instance of the `Has_text` class and so takes a text configuration option. It also takes a listener which is passed the void value `'()'` every time the button is pressed.

We therefore need to make the button talk to the listener provided by the `BVar`. We do this using `tellL`.

```
tellL :: a -> Listener a -> Listener b
```

Figure 21 shows how `tellL` works. It takes a listener expecting values of type `a` and a value of type `a`. It produces a composite listener that, when it is fired, ignores its argument and always performs its action with the given value.

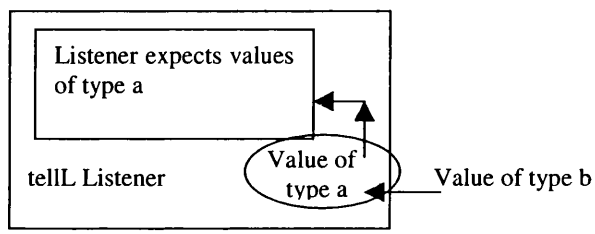

Figure 21 - The `tellL` Listener

In the definition of `incb` we therefore produce a listener that ignores its argument and always updates the `BVar` using the function `inc`.

4.4.5. The complete interface

We create the final interface in two more stages. Firstly, we create the composite component.

```
counterB :: BVar Int -> Component
counterB m =  above (lbl m) (beside (incb m) (decb m))

composite :: BVar Int -> Component
composite m = above (counterB m) (scale m)
```

Finally, we create a BVar to represent the state, and then render the component in a window[6].

```
main :: IO ()
main = display $ mkWindow [] compositeCounter

compositeCounter :: Component
compositeCounter = do {m <- mkBVar 0; composite m}
```

[6] The operator '$' is a Haskell infix operator. It is defined as `f $ x = f x`. In otherwords, it is simply the "apply" operator. It is a way of avoiding parentheses. Without it the function above would have to have been written as `display (mkWindow [] compositeCounter)`.

4.5. The Listener Algebra

FranTk provides an algebra of listener combinators. Though a listener is essentially an imperative callback, this algebra allows us to treat and compose them in a declarative manner. This algebra is dual to the event algebra provided in Fran. Each operation in the event algebra has a corresponding operation in the listener algebra. The choice of operators is therefore based on the set that have proved useful when handling events in FRP. We will first present the most significant operations in the listener algebra, and then define formally how these relate to the event algebra.

4.5.1. The Listener Combinators

The null listener is `neverL`, which does nothing with any value it receives.

```
neverL :: Listener a
```

To merge two listeners we use `mergeL`. This makes a new listener which passes every value consumed to its two argument listeners.

```
mergeL :: Listener a -> Listener a -> Listener a
```

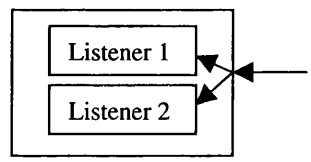

Figure 22 - The `mergeL` Listener

We can therefore define, `allL`, a combinator that merges a list of listeners.

```
allL :: [Listener a] -> Listener a
allL xs = foldr mergeL neverL xs
```

There is a *comap* function on listeners. In contrast to the standard map function, this applies what appears to be an inverse function to a listener. It produces a listener that consumes values, and applies the given function to these values before passing them on to the given listener.

```
comapL :: (b -> a) -> Listener a -> Listener b
```

We can therefore trivially define `tellL` in terms of `comapL`.

```
tellL a l = comapL (const a) l
```

There is a filter function on listeners. This consumes values and passes them on to the given listener, if they satisfy the given predicate.

```
filterL :: (a -> Bool) -> Listener a -> Listener a
```

We can create a one shot listener that consumes one value and then behaves as `neverL` using `onceL`.

```
onceL :: Listener a -> Listener a
```

For instance, we might require a button that could only ever be pressed once. We could define this using `onceL`.

```
mkOnceButton :: [ConfB Button] -> Listener a -> Component
mkOnceButton cs l = mkButton cs (onceL l)
```

We can make a listener snapshot a behavior and consume its current value. For instance, we have `snapshotL`.

```
snapshotL :: Behavior b -> Listener (a,b) -> Listener a
```

As shown in Figure 23, every time the new listener consumes a value it samples the behavior and passes the pair of values to its argument listener.

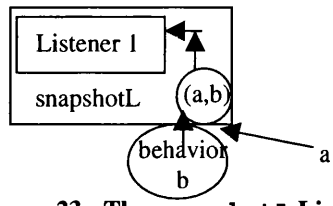

Figure 23 - The `snapshotL` Listener

This is a very useful combinator. It is often necessary to sample the state of the application when some user input occurs. For instance, we may need to check which mode the system is in to interpret the input. As an example, consider a distance converter. It could be represented in terms of two states. The current units and the distance value represented in some neutral, arbitrary units such as Miles. We assume the existence of a function to convert a value and units pair into a distance in Miles.

```
type Units = BVar Unit
type Distance = BVar Double
convert :: (Double,Unit) -> Double
```

To update the distance we would therefore need a listener that sampled the current units and transformed the given value into Miles.

```
inputDistance :: BVar Double -> BVar Unit
              -> Listener Double
inputDistance distBV unitsBV =
 snapshotL (bvarBehavior unitsBV) $
    comapL convert $
          bvarInput distBV
```

There is a listener equivalent of the `scanl` function.

```
scanlL :: (a -> b -> a) -> a -> Listener a -> Listener b
```

This works as shown in Figure 24. The listener's current value starts with the initial value provided. Every time the listener consumes a value `b`, it applies its update function `f` to its current value `a` and the new value, `b`. It passes `(f a b)` to the argument listener, and updates its current value as well. This function is used in Section 4.10.2.

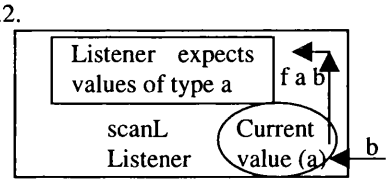

Figure 24 – The `scanlL` Listener

We can define a "listener level switcher". This creates a reactive listener that may itself change with time.

```
switcherL :: Listener a -> Event (Listener a) -> Listener a
```

Consider the listener `switcherL l e`. When this composite listener consumes values, it starts by passing them to the initial listener `l`. Whenever the event `e` fires, it switches to a new listener and passes all values consumed to it.

We can get access to the time that a listener consumes a value using `withTimeL`.

```
withTimeL :: Listener (a,Time) -> Listener a
```

Here the type `Time` is a synonym for `Double` (semantically these time values must be non-negative), and represents the number of seconds since the start of the program.

We can make a listener that simply performs an `IO` action using `mkL`.

```
mkL :: (a -> IO ()) -> Listener a
```

We can also make a listener that performs a GUI action using `mkGUIL`.

```
mkGUIL :: (a -> GUI ()) -> Listener a
```

Composing listeners can be thought of as forming a pipeline through which data will pass. For instance, consider `comapL f $ comapL g $ l`. Values consumed by this composite listener are first processed by `comapL f`; this generates a new value which is passed to `comapL g`; the final result is then passed to `l`.

Previous functional toolkits would require a programmer to write this sort of code in terms of an IO action that sampled a mutable variable. With the listener algebra, programmers are able to manipulate imperative actions and compose them in a declarative style.

4.5.2. Event-Listener Duality

The event and listener algebras in Fran are duals of each other[7]. We can therefore formally define a set of relationships between them. The primitive combination operation for events and listeners is `addListener`. This adds a listener to an event such that the listener is fired every time there is an event occurrence. This function returns a remove action that can be called to unregister the listener's interest.

```
addListener :: Event a -> Listener a -> GUI (GUI ())
```

The relationship between events and listeners can be defined in terms of the `addListener` function. For any `e :: Event a, l :: Listener a, b :: Behavior b, f :: a -> b, p :: a -> Bool, e1 :: Event b, n :: a, op :: a -> b -> a, l1 :: Listener b, l2 :: Listener (a,b)`

```
addListener neverE l <==> addListener e neverL

neverE :: Event a
-- an event that never generates any occurrences

addListener (mapE f e) l1 <==> addListener e (comapL f l1)

mapE :: (a -> b) -> Event a -> Event b
-- map a function over each event occurrence

addListener (filterE p e) l <==> addListener e (filterL p l)

filterE :: (a -> Bool) -> Event a -> Event a
-- filter out event occurrences that don't match the predicate

addListener (onceE e) l <==> addListener e (onceL l)

onceE :: Event a -> Event a
-- yield an event with only one occurrence (the first produced)

addListener (snapshotE b e) l2 <==> addListener e (snapshotL b l2)

snapshotE :: Behavior a -> Event a -> Event (a,b)
-- snapshot a behavior on each event occurs

addListener (scanlE op n e1) l <==> addListener e1 (scanlL op n l)

scanlE :: (a -> b -> a) -> a -> Event b -> Event a
-- accumulate a value in a similar way to scanlL
```

[7] The Event algebra operations are defined in Appendix A.

There are two combinators not included in the list above, `mergeL` and `switcherL`. These both have duals in the event algebra.

```
(.|.),mergeE :: Event a -> Event a -> Event a
switcherE :: Event a -> Event (Event a) -> Event a
```

However, they cannot simply be defined in terms of each other because their types are too different. The combinator `mergeE` takes two events rather than two listeners as arguments; the combinator `switcherE`, creates an event level switcher instead of an listener level switcher.

The event level switcher starts by generating occurrences from the first event. After every event-valued occurrence, the switcher generates occurrences from the new event. Note that we can use the event-level switcher to define a monadic instance for events. This begins by generating no occurrences (that is it behaves like `neverE`). After every occurrence of `e`, it applies `f`, to create a new event, and generates occurrences from this new event[8].

```
(>>) :: Event a -> (a -> Event b) -> Event b
e >>= f = neverE `switcherE` e ==> f
```

This allows us to define sequencing on events. For instance, the "`onlyAfter`" event below will only generate occurrences from `e2`, after `e1` has generated its first occurrence.

```
onlyAfter e1 e2 = onceE e1 >> e2
```

This definition is in fact very similar to the monad of imperative streams used in Pidgets (Section 3.14). The expression `(do {x <- e1;f x})`, means after every occurrence `x`, produced by `e1`, generate occurrences from the new event '`f x`'. The major difference is that events do not support imperative actions explicitly, we must instead add a listener to the new event to get such an effect.

If we imagine a wire connecting a listener to an event, the event and listener combinators can be interpreted as mechanisms to transform user input code at either end of the wire (Figure 25).

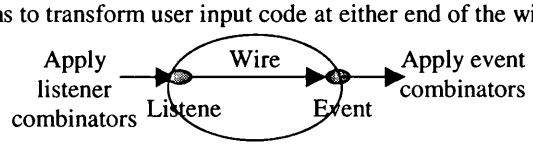

Figure 25 - A FranTk Wire

4.6. Introducing Wires

The BVar is not a FranTk primitive. The communication primitive is, in fact, the wire shown in Figure 25. A `Wire` is more limited than a `BVar`. In particular, it is stateless and has no behavior. It has only an input listener and an event.

```
mkWire :: GUI (Wire a)
newWire :: IO (Wire a)
wireInput :: Wire a -> Listener a
wireEvent :: Wire a -> Event a
```

We can define a BVar in terms of a wire.

```
data BVar a = BVar {
    bvarUpdInput :: Listener (a -> a),
    bvarEvent :: Event a,
    bvarBehavior :: Behavior a
    }
```

[8] The use of quotation marks (eg `` `switcherE` ``) turns a standard Haskell function into an infix function.

```
newBVar :: a -> IO (BVar a)
newBVar a = do
 (l,e) <-  newWire
 let e' = a `accumE` e
 let b = a `stepper` e'
 return (BVar l e' b)
```

The definition of a BVar relies on two Fran combinators. The BVar hears update function values on the wire. These updates will modify its state.

It accumulates an event based value using `accumE`. This will therefore form an event that produces an occurrence on every update, by applying the update function to the current BVar value[9].

```
accumE :: a -> Event (a -> a) -> Event a
```

The function `accumE` is defined in terms of `scanlE`.

```
accumE x0 change =
  scanlE apply x0 change
  where
    apply :: a -> (a -> a) -> a
    apply a f = f a
```

The BVar's behavior is formed by stepping through, changing on every event occurrence.

```
stepper :: a -> Event a -> Behavior a
```

A stepper is in fact a version of a more general reactive behavior combinator, that switches between behaviors on event occurrence.

```
switcherB :: Behavior a -> Event (Behavior a) -> Behavior a

stepper a e = lift0 a `switcher` (e ==> lift0)
```

We can use this approach to define other types of BVar, such as BVar collections discussed in Section 4.8.

It is sometimes also useful to be able to generate a BVar which listens to an input event as well its actual listener. Again such a BVar can easily be defined. For instance, this function was useful in the ATC case study, discussed in section 6.3.4

```
newBVarE :: a -> Event (a -> a) -> IO (BVar a)
newBVarE a inpE = do
 (l,e) <-  newWire
 let e' = a `accumE` (e .|. inpE)
 let b = a `stepper` e'
 return (BVar l e' b)
```

4.7. Simple Dynamic Interfaces

The previous examples have shown how to implement simple interactive systems in FranTk. However, they have involved only a static set of widgets on screen. That is, though the appearance of individual labels has changed, the number of labels has not. The ability to handle dynamically changing collections of components in FranTk is one of its major benefits.

There are two sorts of dynamic display we could have. The first is a simple conditional display. Here we can display one of two components depending on some state. The second sort of dynamism is the introduction of new components on to a screen. We will introduce a conditional display in this section, and then a full dynamic display in Section 4.8.

---

[9] There are, in fact, some problems with the types of these functions (`accumE`, `stepper`) which will be discussed in Section 7.2.5. This issue is also discussed in Appedix A.

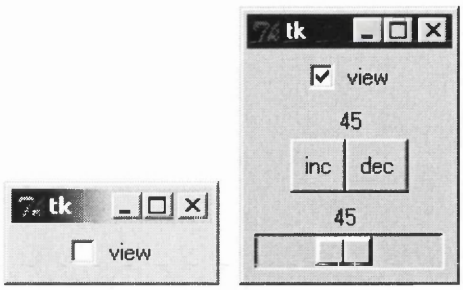

Figure 26 - A Conditional Display in FranTk

Consider the interface shown in Figure 26. It consists of two parts a checkbutton, and a composite counter from Section 4.4. The checkbutton is used to control whether the counter component is visible or not.

4.7.1. Defining a Checkbutton

We create a Boolean `BVar` that models the visibility of the counter component. The checkbutton then talks to this `BVar`. We define this using `mkCheckbutton`.

```
vischeck :: BVar Bool -> Component
vischeck visBv =
 mkCheckbutton [text "view", checkVal True] (bvarInput visBv)

mkCheckbutton :: [Conf Checkbutton] -> Listener Bool -> Component
```

We set its initial state using `checkVal`.

```
checkVal :: Has_checkVal w => Bool -> Conf w

instance Has_checkVal Checkbutton
```

4.7.2. Conditional displays

We can conditionally display a component using `ifB`.

```
condCounter :: Component
condCounter = do
  visBv <- mkBVar True
  above (vischeck visBv)
        (ifB (bvarBehavior visBv) compositeCounter emptyComponent)

class GBehavior w where
  ifB :: GBehavior w => Behavior Bool -> w -> w -> w

instance GBehavior Component
```

When applied to components `ifB b w1 w2` produces a component which behaves as `w1` when `b` is True and `w2` otherwise. (Other members of the `GBehavior` class include `Behaviors` and `Events`.) In this example we display the `compositeCounter` when the BVar has the value `True` and an empty component otherwise.

```
emptyComponent :: Component
```

This provides us with our first mechanism for dynamically altering the number of widgets on screen at any given time.

4.8. Displaying Dynamic Collections

In the previous section, we discussed interfaces with conditional displays. Now we will consider interfaces with a truly variable number of widgets on screen at any given time. For instance, consider the interface in Figure 27. It shows a collection of the conditional displays defined in Section 4.7. We can add new components to the bottom of the window by pressing the *Create* button.

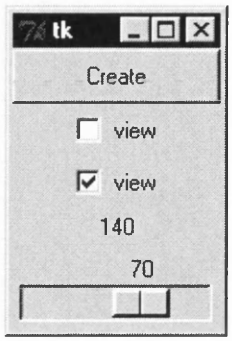

Figure 27 - A Dynamic Interface in FranTk

We need to define the collection of objects that are displayed on the screen. In most previous GUI systems, we would do this by performing update actions that add and delete widgets from the screen. In FranTk we define the appearance of an interface based on some state for all time. We therefore need to be able to define the user's view as a function of some abstract collection type. We do this using a behavioral collection, in this case a list.

```
type ListB a

nabove :: ListB Component -> Component
```

In FranTk we represent a dynamic list of objects as a `ListB`. To geometrically compose dynamic collections, we use combinators such as `nabove`, which places a dynamic collection of components above each other. We can think of a `ListB` as a behavior list (`Behavior [a]`). However, when rendered the `ListB` will incrementally update the screen only making necessary changes, rather than redisplaying everything.

To make a list collection we use a special type of `BVar`, a `ListBVar`. We give it an initial list of values.

```
type ListBVar a
mkListBVar :: [a] -> GUI (ListBVar a)
newListBVar :: [a] -> IO (ListBVar a)
```

We extract the dynamic list from the variable using `collection`.

```
collection :: ListBVar a -> ListB a
```

We can append items to a `ListBVar` using `appendListB`.

```
appendListB :: ListBVar a -> Listener a
```

To implement our example, we would therefore have the following code.

```
dynamicExample :: Component
dynamicExample = do
  lst <- mkListBVar []
  let create = mkButton [text "Create"]
                        (tellL condCounter (appendListB 1))
  above create (nabove (collection lst))
```

4.9. Dynamic Collections

As we saw in the previous section, behavioral collections allow us to model dynamic collections of objects and treat them as behaviors. They can, however, also be incrementally rendered onto a display, so that only changes to the collection are redrawn. Dynamic collections such as sets and lists are defined in terms of a general behavioral collection type.

```
data CollB entry op c a
```

The type is parameterised over its collection type, `c`, element type, `a`, and update operations, *op*, and internal structure, `entry`. It models a static collection of type `c a`. It is altered by incremental updates of type `op a`. It maintains internal data using the type `entry a`. For the purposes of this Chapter, only the last two type parameters are important. The others will be explained when discussing the implementation in Section 7.5.

4.9.1. List Collections

Using the generic CollB type, dynamic Lists are defined as shown below. They model a list of values, and have a corresponding update and internal entry type.

```
type ListB a = CollB Entry ListUpd [] a

data Entry a
data ListUpd a
```

We can get a `Behavior` from a `ListB`.

```
listBehavior :: ListB a -> Behavior [a]
```

This allows us to treat `ListB` values as normal behaviors when convenient. For instance, we could lift standard list functions and apply them to the list behavior, such as defining a lifted size function that returns the length of the list for all time.

```
sizeB :: ListB a -> Behavior Int
sizeB l = lift1 size (listBehavior l)
```

We can therefore define lifted versions of all the standard list observer functions.

4.9.1.1. Creating list collections

We can create a `ListB` from an initial list and an update event. It begins by behaving as the initial list, and then on every event occurrence, changes by applying the update function from the occurrence.

```
mkListB :: IList a -> Event (IList a -> IList a) -> ListB a
data IList a
```

This is therefore similar to the Fran behavior combinator, `stepAccum`, which creates a piecewise constant behavior that is updated by event occurrences[10].

```
stepAccum :: a -> Event (a -> a) -> Event a
```

The `IList` type is a special incremental list type that maintains incremental updates. The Haskell Edison library [145], defines a general interface for dealing with functional data structures such as `Sequences` and `Sets`. The `IList` type implements the `Sequence` interface, allowing us to treat

[10] The function `stepAccum` is based on `stepper`. There is therefore also a problem with the type of this function (there is a similar problem with `mkListB`). See Section 7.2.5, Section 7.5.2 and Appendix A for more on this.

them in the same way as standard lists. This, therefore, provides a powerful, and familiar set of operators for constructing dynamic lists.

```
instance Sequence IList
```

We can, therefore, generate `ILists` using the standard `Sequence` constructors.

```
empty :: IList a
single :: a -> IList a
fromList :: [a] -> IList a

cons :: a -> IList a -> IList a -- add an element to the front
snoc :: a -> IList a -> IList a -- add an element to the back
append :: IList a -> IList a -> IList a
```

We can also apply standard list functions such as `size`, `map` and `fold` to the list.

4.9.1.2. Using dynamic lists

The dynamic list type implements many of the standard list functions. For instance, we can construct a constant list behavior from a static list. We can also append dynamic lists, remove duplicate elements from them (using nub), reverse them, map functions along them, filter them, partition them and sort them.

```
fromList :: [a] -> ListB a
append :: ListB a -> ListB a -> ListB a
nub :: Eq a => ListB a -> ListB a
reverse :: ListB a -> ListB a
map :: (a -> b) -> ListB a -> ListB b

filter :: (a -> Bool) -> ListB a -> ListB a
partition :: (a -> Bool) -> ListB a -> (ListB a,ListB a)

sort :: Ord a => ListB a -> ListB a
sortBy :: (a -> a -> Ordering) -> ListB a -> ListB a
```

It is also helpful to be able to apply behavior-based functions to them. For instance, we might wish to take a changing number of elements from the front a list. We can do this using `takeB`, which takes an `Int` behavior, instead of a simple static integer as its first argument. As an example of use, consider an interface which should display only the first 'n' elements of some dynamic list where 'n' was controlled by a user via a slider.

```
takeB :: Behavior Int -> ListB a -> ListB a
```

We might also wish to `sort`, `partition` or `filter` a list based on some behavior based function.

```
filterB :: (a -> Behavior b) -> Behavior (b -> Bool) -> ListB a
        -> ListB a

partitionB :: (a -> Behavior b) -> Behavior (b -> Bool) -> ListB a
           -> (ListB a,ListB a)

sortByB :: (a -> Behavior b) -> Behavior (b -> b -> Ordering)
        -> ListB a -> ListB a
```

These combinators each take a function to extract a behavior from a list element, and a function valued behavior to apply. We need this flexibility, because the elements of a dynamic list may themselves be dynamic. For instance, imagine that we were maintaining a collection of counter components, each with a name and a BVar representing the state of the counter.

```
data CounterElt = CounterElt {name :: String, value :: BVar Int}
type Counters = ListB CounterElt
```

We might wish to display a list of sorted labels and values. The list could be sorted into either ascending or descending order, based on some Boolean behavior and the current value of the counter.

```
view :: Behavior Bool -> ListB CounterElt -> Component
view sortrule lst =
  nabove (fmap viewElt
           (sortByB (bvarBehavior . value) (lift1 sorter sortrule)
                    lst))
 where
   sorter True = (>)
   sorter False = (<)

   viewElt (nm val) = mkLabel [textB (lift1 (nm ++ " " ++) val)]
```

We therefore have the ability to treat dynamic lists as simple Haskell values, and manipulate them in a powerful, declarative manner. We can use the `ListB` type when placing windows or widgets, or to make menus, listboxes and text areas displaying dynamic data.

4.9.2. Set Collections

A dynamic set can be defined in terms of the general collection type as follows. It models a set of values, with corresponding update and internal entry data.

```
type SetB a = CollB Entry SetUpd Set a
data SetUpd a
data Entry a
```

Again we can extract a behavior from a dynamic Set, in order to observe its current state.

```
setBehavior :: SetB a -> Behavior (Set a)
```

We create a dynamic Set using an initial static set and an event generating update functions[11].

```
mkSetB :: ISet a -> Event (ISet a -> ISet a) -> SetB a
```

Here the `ISet` type implements Edison's Set interface [145]. We can therefore, for instance, create sets, insert, delete, filter, partition, and construct the union, intersection and difference of `ISets`.

The `SetB` type implements many of the standard Edison `Set` operations [145]. For instance, we can create constant dynamic sets from lists and filter and partition them. We can also form the union, intersection and difference of two dynamic sets.

```
fromList :: [a] -> SetB a
filter :: (a -> Bool) -> SetB a -> SetB a
partition :: (a -> Bool) -> SetB a -> (SetB a,SetB a)
union, intersect, difference :: Eq a => SetB a -> SetB a -> SetB a
```

Finally, we can apply behavior valued filter and partition functions to a dynamic set, using a similar interface to dynamic lists.

```
filterB :: (a -> Behavior b) -> Behavior (b -> Bool) -> SetB a
        -> SetB a
partitionB :: (a -> Behavior b) -> Behavior (b -> Bool) -> SetB a
           -> (SetB a ,SetB a)
```

4.9.3. Bag collections

We sometimes need to generate a collection with no initial notion of equality. We might, however, still be able to define some predicate when inserting an item, that specifies when it should be deleted. For this, we can use a dynamic bag collection.

---

[11] As with `mkListB` there is again a problem with the type of this function.

```
type BagB a = CollB BagEntry BagUpd Bag a
data IBag a

mkBagB :: IBag a -> Event (IBag a -> IBag a) -> BagB a[12]
```

When adding an element we also pass in an event that will generate one occurrence when the item is to be deleted.

```
insert :: a -> Event () -> IBag a -> IBag a
fromList :: [(a,Event ())] -> IBag a -> IBag a
```

This approach, though usually unnecessary, can sometimes be very useful. An example of its use is shown in Section 4.12.2.

4.9.4. Collection variables

As with standard behaviors, it is useful to have behavior collection variables. We can define a generic behavior variable, `GenBVar`. This maintains a value, and a listener that expects update functions. In FranTk, as we generally create behaviors using Behavior variables, it is useful to have one generic type that can be reused.

```
data GenBVar a coll = GenBVar (Listener (a -> a)) coll
```

We have standard functions to extract the *value, input listener*, and *update-input listener* from a generic behavior variable.

```
collection :: GenBVar a coll -> coll
collection (GenBVar _ c) = c

bvUpdInput :: GenBVar a coll -> Listener (a -> a)
bvUpdInput (GenBVar l _) = l

bvInput :: GenBVar a coll -> Listener a
bvInput (GenBVar l _) = comapL const l
```

For a given dynamic collection we can also extract a behavior, representing its state for all time. As an example, consider the list behavior variable. This maintains a list collection and `IList` valued updates.

```
type ListBVar a = GenBVar (IList a) (ListB a)

lbvarBehavior :: ListBVar a -> Behavior (c a)
lbvarBehavior c = listBehavior (collection c)
```

We can now define the `ListBVar` operations used in Section 4.8. We create a `ListBVar` by creating a wire and then generating a `ListB` using the event stream from the wire.

```
mkListBVar :: [a] -> GUI (ListBVar a)
mkListBVar as = do
  w <- mkWire
  let lst = mkListB (fromList as) (wireEvent w)
  return (GenBVar (wireInput w) lst)
```

We can define the `appendListB` listener, in terms of the update listener and the `IList snoc` combinator, which adds an element to the end of a list.

```
appendListB :: ListBVar a -> Listener a
appendListB l = comapL snoc (cbvarInput lst)
```

We can also define basic BVars in terms of these generic BVars.

---

[12] As with `mkListB` there is again a problem with the type of this function.

```
type BVar a = GenBVar a (Behavior a,Event a)

bvarUpdInput :: BVar a -> Listener (a -> a)
bvarUpdInput (GenBVar l _) = l

bvarInput :: BVar a -> Listener a
bvarInput (GenBVar l _) = comapL const l

bvarBehavior :: BVar a -> Behavior a
bvarBehavior (GenBVar _ (b,_)) = b

bvarEvent :: BVar a -> Event a
bvarEvent (GenBVar _ (_,e)) = e
```

4.9.5. Simplifying the name space

We now have a number of data types that possess behaviors, events or listener updates. Unfortunately, we have a large number of functions to extract data from each of these types. Yet these extraction functions are similar for each data type. We can use Haskell type classes to simplify the name space.

We can define a class, `HasBehavior`, for any object that has a behavior. This relies on *functional dependencies* to work [104]. The result type `a`, must be extractable from the argument type `c`.

```
class HasBehavior c a | c -> a where
 behavior :: c -> Behavior a
```

We therefore have instances of this for BVars, dynamic collections and dynamic collection variables.

```
instance HasBehavior (BVar a) a
instance HasBehavior (CollectionB f b c a) (c a)
instance HasBehavior (GenBVar a (CollectionB f u c a)) (c a)
```

This provides us, for instance, with a behavior function extracting a behavior list from a `ListB`.

```
behavior :: ListB a -> Behavior [a]
```

Similarly, we can define a class `HasEvent` for any object that has an event. In this case, we have instances for BVars and Wires.

```
class HasEvent c w | c -> w where
   event :: c -> Event w

instance HasEvent (BVar a) a
instance HasEvent (Wire a) a
```

We can define a class, `HasInput`, for objects with an input listener, and a class `HasUpdInput`, for objects with an update input listener.

```
class HasInput c w | c -> w where
 input :: c -> Listener w

class HasUpdInput c w | c -> w where
 updInput :: c -> Listener (w -> w)
```

Wires have only input listeners. Standard and collection BVars possess both input and update input listeners.

```
instance HasInput (Wire a) a
instance HasInput (GenBVar a coll) a

instance HasUpdInput (GenBVar a coll) a
```

`ListBVars`, for instance, have an update listener that expects `IList` based updates.

```
updInput :: ListBVar a -> Listener (IList a -> IList a)
```

The use of type classes here has allowed us to significantly simplify the name space, so that we have only four basic operators, `behavior`, `event`, `input` and `updInput`.

Using the generic dynamic collection type we can easily build a range of specific collection types, making the programming style very reusable.

4.10. Adding Windows and Menus

In FranTk, we can easily construct windows with menus, again including menus with dynamic numbers of components. For instance, consider the interface in Figure 28. It displays a dynamic list of labels, each with a unique integer value. We can add new items and delete existing items. We can also copy the window, to create a new window that is a view on to the same collection. Each window has a menu bar with one cascading menu. This offers three options, an "add" item button, a "copy window" button and a "delete" button that displays a cascading menu with a delete button for every currently active item.

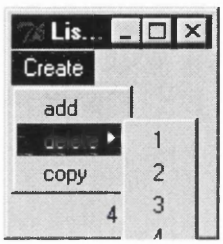

Figure 28 - Windows and Menus

4.10.1. Creating a Window

We define the window as follows.

```
window :: Listener () -> ListBVar Int -> WComponent
window addWindow lst =
 mkWindow [useMenu (createMenu addWindow lst)]
          (nabove (fmap label (collection lst))
  where label n = mkLabel [text (show n)]

mkWindow :: [Conf Window] -> Component -> WComponent

useMenu :: GUI Menu -> Conf Window
```

We create a window with a given menu (added with `useMenu`), displaying a label for each item in the list. The ability to use one simple model of the data (a `ListBVar`) and provide multiple views using functions such as `nabove` is very significant. We have a very expressive and succinct way of describing interface components.

4.10.2. Creating the Menus

We now create the menus required for the example. We create a menu with either a static, or a dynamic, list of menu items.

```
mkMenu :: [Conf Menu] -> [MenuItem]  -> GUI Menu

mkMenuL :: [Conf Menu] -> ListB MenuItem -> GUI Menu
```

In our example, we use two different types of menu item, cascading menus and menu buttons.

```
mcascade :: [Conf Cascade] -> GUI Menu -> MenuItem
mbutton :: [Conf MButton] -> Listener () -> MenuItem
```

The "create" menu consists of two items an "add" button and a cascading "delete" menu, with a button for every current item.

We define the delete cascading menu as follows.

```
deletemenu :: ListBVar Int -> MenuItem
deletemenu lst =
 mcascade [text "delete"] (mkMenuL [] (fmap item (collection lst))
 where
  item :: Int -> MenuItem
  item n = mbutton [text (show n)]
                   (tellL (delete n) (updInput lst))
```

The cascading menu consists of a list of numbered buttons, one for each active component in the dynamic list.

We define the "add" menu item as follows.

```
addmenu :: ListBVar Int -> MenuItem
addmenu lst =
  mbutton [text "add"] (scanlL inc 0 $ comapL snoc $ updInput lst)
   where
     inc :: Int -> a -> Int
     inc n _ = n + 1
```

This uses `scanlL` to accumulate new unique labels as it goes along. Every time the menubutton is pressed it adds a new item to the list with a value one higher than the last item added.

The copy menu item simply fires the add-window listener when pressed.

```
copymenu :: Listener () -> MenuItem
copymenu l = mbutton [text "copy"] l
```

We can now define the menu bar.

```
createMenu :: Listener () -> ListBVar Int -> GUI Menu
createMenu l lst =
  mkMenu [] [mcascade [text "Create"]
              (mkMenu [] [addmenu lst, deletemenu lst,copymenu l])
            ]
```

4.10.3. Creating new window instances

To complete the example, we need to generate new windows every time a button is pressed. There are two possible ways to do this. The simplest and least powerful mechanism is just to create a listener that renders a new window every time it is fired. We render one initial window to start with.

```
runWindows :: ListBVar Int -> GUI ()
runWindows l = do
 let add = mkGUIL (render (window add l))
 render (window add l)
```

However, we might wish to use and display a dynamic collection of windows. This would be important if we had some abstract model of the set of windows on screen. We can create a dynamic list, containing the lists to be displayed. To add a new window, we just append a new item to this list. We then apply the window function to each element, to generate a dynamic list of windows, place them (using `pile`) and then render them.

```
runWindows :: ListBVar Int -> GUI ()
runWindows l = do
  windows <- mkListBVar [l]
  let add = tellL (snoc l) (updInput windows)
  render (pile (fmap (window add) windows)
```

```
class Pile c w where
  pile :: c w -> w

instance Pile ListB WComponent
```

FranTk therefore provides a flexible, powerful approach to allow the creation of multi-window programs, containing dynamic menus.

4.11. Selectable Components

We need to be able to manipulate selectable components in FranTk. For instance, consider the following interface. It contains two buttons and a listbox. Pressing the *Create* button adds items to the listbox. Pressing the *Delete* button deletes the currently selected item.

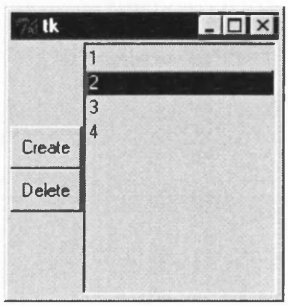

Figure 29 - Using a Listbox

We can implement this as shown below.

```
listcomponent :: ListBVar Int -> Component
listcomponent lst = do
 (selectedBv :: BVar [String]) <- mkBVar []
 let create = mkButton [text "Create"]
                  (scanlL inc 0 $ comapL snoc $ updInput lst)
     del = mkButton [text "Delete"]
                  (snapshotL_ (behavior selectedBv) $
                      comapL deletes $
                          updInput lst)
     lstbx = mkListbox [listItemsLB (fmap show (collection lst)),
                        listenSelection (input selectedBv)]
 beside (above create del) lstbx

mkListbox :: [Conf Listbox] -> Component
listItemsLB :: ListB String -> Conf Listbox
```

We have a *Create* button that generates new items, with a unique name. It operates similarly to the *Create* menu button from Section 4.10.2. We use a BVar to store the selection state of the listbox. This maintains a list of the String values of the selected items. When the *Delete* button is pressed, we sample the selection state using `snapshotL_`. We then update the list by deleting all the selected items, using the `deletes` function. It is important to note that though this presents a different view from the menu based one in the previous section, the underlying application data structure is identical: both simply use `ListBVars`.

We create a listbox that displays all the items in the dynamic list, using `listItemsLB`. We tell the selection BVar about all changes using the `listenSelection` option.

In general, any component that supports selection will implement the following two interfaces, `HasSetSelection` and `HasGetSelection`. Note that these are overloaded on both the component and the index type.

We can set the selection indices using the `HasSetSelection` class. This allows the selection to be set initially, or to be updated by an event or to be based on a behavior.

```
class HasSetSelection w i where
  setSelection :: i -> Conf w
  setSelectionE :: Event i -> Conf w
  setSelectionB :: Behavior i -> Conf w
```

We can access the selection in one of two ways. The first is the most obvious. We can add a listener to the component that is told when the selection changes. This is the style we used in the example above.

```
class HasGetSelection w i where
  listenSelection :: Listener i -> Conf w

  snapSelection :: Event a -> Listener (a,i) -> Conf w
```

The second approach is slightly more complex but sometimes useful. We can pass an event and a listener to the component. Every time there is an event occurrence, the selection state is sampled and is passed on to the listener.

We could rewrite the example above to use this approach.

```
listcomponent :: ListBVar Int -> Component
listcomponent lst = do
 w <- mkWire
 let mk = ... -- unchanged
     del = mkButton [text "Delete"] (input w)
     lstbx = mkListbox
               [listItemsLB (fmap show (collection lst)),
                snapSelection (event w)
                        (comapL deletes (updInput lst))]
 beside … -- unchanged
```

We create a wire that the *Delete* button talks to. The listbox, listens to the wire, samples its selection and tells the `ListBVar` to delete the selected items. This therefore has the structure shown in Figure 30.

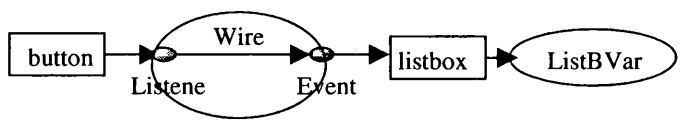

Figure 30 - Using the `snapIndex` Function

This approach is therefore more complex. However, it has one important advantage over the *listenIndex* based approach. If the selection state changes regularly, but is only sampled very occasionally, this approach will be significantly more efficient. This is because we only perform work when we need to sample the selection state, rather than every time it is changed.

If a component supports both the set and get interfaces for a given index type, then we can bind a BVar to its selection state. This BVar will be updated every time the selection is changed by the user, and will update the selection every time its state is changed.

```
useSelection :: (HasSetSelection w i,HasGetSelection w I) =>
                  BVar i -> Conf w
```

Listboxes support the set and get interfaces for three types of index. Selection indices can be accessed by their position in the listbox, or by the list item String, or by a special index type (`Loc`) that supports positions and a tag (`IEnd`) meaning the last item in the list. When setting the selection using the String form, every item with the String name will be selected.

```
instance HasGetSelection Listbox [Int]
instance HasSetSelection Listbox [Int]
instance HasGetSelection Listbox [String]
instance HasSetSelection Listbox [String]
```

```
instance HasGetSelection Listbox (Maybe (Loc,Loc))
instance HasSetSelection Listbox (Maybe (Loc,Loc))
data Loc = I Int | IEnd
```

FranTk therefore provides good support for selectable components. The use of behaviors and listeners is again very powerful here. As with most model/view approaches it would be easy, for instance, to create two listboxes that maintained the same selection state (by making them share the same selection BVar).

However, the FranTk approach is more powerful than most model/view approaches, as we can easily define values that are functions of several behaviors. For instance, we can provide a simple implementation of two listboxes whose selection states are mutually exclusive. That is, any item selected in the first listbox will be unselected in the second, and vice versa.

```
exclusive :: ListBVar Int -> Component
exclusive lst = do
  select <- mkBVar ([],[])
  let lb set get =
        mkListbox [listItemsLB (collection lst),
                   listenSelection (comapL set (updInput select)),
                   setSelectionB (lift2 get (behavior select)
                                            (behavior lst))]
  beside (lb set1 get1) (lb set2 get2)

set1,set2 :: [String] -> ([String],[String])-> ([String],[String])
set1 xs (s1,s2) = (xs,s2 \\ xs)
set2 xs (s1,s2) = (s1 \\ xs,xs)

get1,get2 :: ([String],[String]) -> [String] -> [String]
get1 (s1,s2) all = s1
get2 (s1,s2) all = s2 `union` (all \\ s1)
```

We model the selection state using a BVar with a pair of String lists, one for each listbox. We create two listboxes that both use and update this BVar. When we set the selection state in one component, we unselect all those items for the other component (the \\ operator deletes one list from another). Significantly, the selection view is a function both of the selection state and the current list. There may be elements that have been added and not explicitly selected in either listbox. However, to maintain the selection invariant, they must be implicitly selected in one. To guarantee the selection invariant, the second listbox selects all items explicitly selected in its box, and all items that have not been selected in the first box. We can therefore maintain a complex invariant in a few simple lines of code.

4.12. Dynamic Animations

FranTk, like Fran, also provides support for graphical animations. For instance, consider the animated ball presented in Section 3.13.1. We will now see how to implement it in FranTk.

4.12.1. Introducing Canvases

We can compose standard components using geometric combinators such as *above* and *beside*. However, to produce graphical animations we need to be able to compose pictures at arbitrary locations on a screen. We can do this using a Canvas component.

```
mkCanvas :: [Conf Canvas] -> CComponent -> Component
```

A `Canvas` is a widget that may contain canvas items. It contains a `CComponent`. A `CComponent` is an action that produces a `CanvasItem`.

```
type CComponent = GUI CanvasItem
```

This definition allows canvas items to have local state which is useful in our animated ball example.

```
movingball :: CComponent
movingball = do
  time <- timeTickNow 0.1
  moveXY 0 (sin time) (mkCOval (vector2XY 30 30) [fill red])

timeTickNow :: Time -> GUI (Behavior Time)
```

The `timeTickNow` operates like the `timeTick` function, introduced in Section 4.3.2, except that the time behavior represents the time since the action was performed (rather than the time since the beginning of the program).

4.12.2. A Dynamic Set of Moving Balls

In FranTk, we can also easily animate a dynamically changing collection of moving balls. For instance, consider the following example. Every time the user clicks the mouse, a ball will be created. This will follow the sine wave motion and will be deleted when it leaves the screen.

We first define a single bouncing ball. Its motion depends on a time valued behavior. It starts at a given position, p, and moves across the screen. As the time behavior represents the interval since the animation started, we offset the move by the time the ball was created. We define a predicate event, that generates one occurrence when the ball moves off the right hand side of the screen.

```
movingball :: TimeB -> (Point2,Time) -> (CComponent,Event ())
movingball time (p,t) =
  let loc = (p .+^ (point2XY 0 (sin (time - lift0 t))))
  in
  (move loc (mkCOval (vector2XY 30 30) [fill red]),
   onceE (predicate ((xval loc) >* lift0 canvaswidth)))
```

To create the canvas we make a time behavior, that ticks every 100 milliseconds. We create a canvas that contains the moving balls. To model the moving balls we use a dynamic bag collection. This relies on the insert function, defined in section 4.9.3. We pile the bag elements on the screen using the pile function.

```
canvas :: Component
canvas = do
 bg <- mkBagBVar empty
 time <- tickTimeNow 0.1

 let addBall :: (Point2,Time)
             -> (IBag CComponent -> IBag CComponent)
     addBall p b = let (c,e) = movingball time p
                   in insert c e b

 mousePressWithLoc 1 (withTimeL $
                        comapL addBall $
                           updInput w) $

    mkCanvas [width canvaswidth,height canvasheight]
             (pile (collection bg))

class Pile c w where
 pile :: c w -> w

instance Pile BagB CComponent
```

We listen to mouse presses (on mouse button 1) on the canvas using `mousePressWithLoc`, and tell the wire about these presses.

```
mousePressWithLoc :: Has_Input w=> Int -> Listener Point2 -> w-> w
```

This operation is a member of the `Has_Input` class. This class provides a range of functions to listen to user input such as key presses and mouse movement. Every interaction component type is a member of the `Has_Input` class, allowing us to bind user input to any of them.

```
instance Has_Input Component , WComponent, CComponent
```

This is a very powerful mechanism. There is no difference between listening to user input on a primitive or composite object, such as a collection of components. It makes the FranTk programming style very compositional.

FranTk therefore allows the generation of complex animations in the same way as Fran does. However, because animation objects can have local state, and we can listen to user input at any level, we can develop a much wider range of possible programs.

4.13. Text Edits - A Document/View Architecture

In FranTk we have two different kinds of text edit widget; a simple single line text entry, and a complex multi-line editor that supports powerful notions such as hypertext tags.

4.13.1. Single-Line text entries

The single line entry has a very simple interface.

```
mkEntry :: [Conf Entry] -> Component
```

It supports the selection interface, discussed in Section 4.11. We can either get the currently selected text, or selection co-ordinates. We can set the selection using the co-ordinates.

```
instance HasGetSelection Entry String
instance HasGetSelection Entry (Maybe (Int,Int))
instance HasSetSelection Entry (Maybe (Int,Int))
instance HasGetSelection Entry (Maybe (Loc,Loc))
instance HasSetSelection Entry (Maybe (Loc,Loc))
data Loc = I Int | IEnd
```

It also supports the cursor interface. This interface operates identically to the selection interface, offering options to set the cursor with an index, and options to listen to changes or to snapshot the cursor position.

Unlike any of the components we have seen so far, the contents of a text entry can be changed directly by user input as well as by any application code. We therefore need to be able to access the current state of a text entry. We do this using a mechanism similar to the `HasGetSelection` interface. We can sample the contents of the text entry using an event and a listener.

```
snapEntry :: Event a -> Listener (a,String) -> Conf Entry
```

We can therefore easily define, for instance, a text entry widget that tells its state to an input listener every time the return key is pressed. This makes use of `keyPress` function that provides access to any keyboard input. The component communicates via a wire, to sample the entry's state every time the `Return` key is pressed.

```
mkEntryRtrn :: [Conf Entry] -> Listener String -> Component
mkEntryRtrn cs l = do
 w <- mkWire
 keyPress Return (input w) $ mkEntry ((snapEntry (event w) l):cs)
keyPress :: Has_Input w => Key -> Listener () -> w -> w
```

We can also use a dynamic list (`ListB`) to model the state of a text Entry. We can therefore listen to `IList` updates on the text entry, and set its state to view a dynamic list. This new `textLB`

configuration option can be used in addition to the standard `text`, and `textB` options. This approach allows us to easily define several component with views on to the same state.

```
listenEntry :: Listener (IList Char -> IList Char) -> Conf Entry
textLB :: ListB Char -> Conf Entry
```

4.13.2. Multi-Line Text Editors

FranTk supports full-blown multi-line text edit widgets. Edit widgets support the standard selection and cursor interfaces. Locations are referred to via a pair consisting of the line and column number.

```
mkEdit :: [Conf Edit] -> Component

instance HasGetSelection Edit (Maybe ((Int,Int),(Int,Int)))
instance HasSetSelection Edit (Maybe ((Int,Int),(Int,Int)))

instance HasGetCursor Edit (Maybe (Int,Int))
instance HasSetCursor Edit (Maybe (Int,Int))
```

Again we wish to be able to model the state of an edit component abstractly, to allow multiple views of the same state. We might expect to again do this using a dynamic list of characters. However, there are two important reasons why this is not practical.

1. Efficiency - Performing arbitrary updates to a large text buffer, modelled simply as a dynamic list would be very inefficient.
2. Expressiveness - Multi-line text editors provide a range of extra functionality, such as hypertext tags. We need to be able to model the insertion of these tags, along with simple text.

FranTk therefore provides a new dynamic document type, which can be used to model the state of an editor. In keeping with the standard dynamic collections interface, FranTk provides a mechanism to produce document behaviors in terms of a static document type (`IDoc`) and incremental updates; or via a document behavior variable[13].

```
data DocumentB
data IDoc

mkDocumentB :: IDoc -> Event (IDoc -> IDoc) -> DocumentB

data DocumentBVar
mkDocumentBVar :: IDoc -> GUI DocumentBVar
```

Using this type we can therefore easily produce a shared text editor with two windows that edit the same document, as shown in Figure 31.

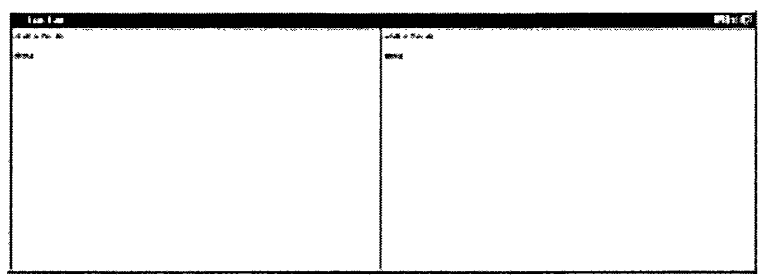

Figure 31 - A Shared Text Editor in FranTk

The definition consists of two edit widgets that use the same document behavior variable.

[13] As with `mkListB` there is again a problem with the type of this function. See Section 8.1.4 for more on this.

```
sharedEditor :: Component
sharedEditor = do doc <- mkDocumentBVar empty
                  let edit = mkEdit [useDocBVar doc]
                  beside edit edit

useDocBVar :: DocumentBVar -> Conf Edit
```

The `useDocBVar` function is actually defined in terms of two more primitive functions.

```
useDocBVar docbv = composeConf (listenDoc (updInput docbv))
                               (setDocB (docB docbv))

composeConf :: Conf w -> Conf w -> Conf w

updInput :: DocumentBVar -> Listener (IDoc -> IDoc)
docB :: DocumentBVar -> DocumentB

listenDoc :: Listener (IDoc -> IDoc) -> Conf Edit
setDocB :: DocumentB -> Conf Edit
```

The `composeConf` function combines two configuration options to form a composite configuration option. We can extract the document behavior using `docB`, and access the update listener using `updInput`. We can listen to all document updates on an edit widget using `listenDoc`. We can set the edit widget to display a document behavior using `setDocB`.

It is also sometimes useful to be able to make an edit widget display an initial `IDoc` or `IDoc` valued behavior. These are helpful when we do not need the full power of the document behavior interface.

```
setIDocB :: Behavior IDoc -> Conf Edit
```

4.13.3. Document Updates

The document update interface, that allows incremental changes to a document to be programmed, is simple but very powerful. There are two basic `IDoc` constructors: we can create an empty document, or a document containing a String.

```
empty :: IDoc
fromString :: String -> IDoc
```

We can insert a String at a given index position, or delete everything between two index positions.

```
insert :: String -> TIndex -> IDoc -> IDoc
delete :: TIndex -> TIndex -> IDoc -> IDoc
```

FranTk also supports the use of structured text. We can construct a document from structured text, or insert some structured at a given index. We can also delete structured text by referring to its name, or replace one named section of structured text with another.

```
structured :: Structured -> IDoc
insertStructured :: Structured -> TIndex -> IDoc -> IDoc
replaceStructured :: Structured -> Ident -> IDoc -> IDoc
```

Structured text consists of standard text, `EditTags` and `EditMarks`. The latter two objects are present in Tcl-Tk. An `EditMark` is an object that marks a particular location in a document; an `EditTag` is an object that covers a section of a text in a document, and can be used to implement hyperlinks and alternative displays. For instance, we could change the font of a single sentence within a text area. Using the structured type we can define nested tags, containing further structured text. The use of structured text allows us to include tags and marks using a simple, declarative style. We can group a section of structured text and give it a particular name, to allow deletion later on.

```
data Structured = SText String
                | STag EditTag
```

```
    | SGroup [Structured]
    | SNamed Structured Ident
    | STextTagged Structured EditTag
    | SMark EditMark
```

To define a given point in an edit area we use the `TIndex` type.

```
data TIndex
```

This has constructor functions to define a point in terms of a line and column number. It also allows us to refer to the start or end of the text, to a particular mark, to the start or end of a particular tag, or to an offset from a given index.

```
tindex :: Int -> Int -> TIndex
tindexEnd :: TIndex
tindexStart :: TIndex
tindexMark :: MrkIdent -> TIndex
tindexTagFirst :: TgIdent -> TIndex
tindexTagLast :: TgIdent -> TIndex
tindexModMove :: TIndex -> ModMove -> TIndex
```

There are also a range of possible offsets, such as to the beginning or end of a line or word.

```
data ModMove = LineStart | LineEnd | WordStart | WordEnd | ...
```

Note that tags and marks are referred to using unique identifiers of type `TgIdent`, and `MrkIdent` respectively. We can refer to the current selection and current insertion cursor using the tag and mark interface.

```
selectTag :: TgIdent, cursorMrk :: MrkIdent
```

4.13.4. Edit Tags – Hypertext Support

An `EditTag` can be used to alter the attributes or, or bind user input to a particular selection of text. We can make a tag using `mkEditTag`.

```
mkEditTag :: [Conf EditTag] -> GUI EditTag
```

The applicable configuration options consist of things like text colour and font.

We can give an `EditTag` a particular unique identifier with `useTgIdent`.

```
useTgIdent :: TgIdent -> Conf w
```

It is this identifier that is referred to in the `TIndex` type.
We can access any user input on an edit tag. It is therefore a member of the `Has_Input` class.

```
instance Has_Input (GUI EditTag)
```

An edit tag implements the selection interface. The selection in question defines area that the tag covers.

```
instance HasGetSelection EditTag (Maybe (Int,Int))
instance HasSetSelection EditTag (Maybe (Int,Int))
```

As an example of the power of this interface, consider the hypertext viewer shown in Figure 32. The ability to support hypertext is very important. Hypertext browsers are currently very common. Since the almost universal acceptance of the web browser, hyperlinks are becoming a common feature of user interfaces. Text editing systems such as Microsoft Word also support them.

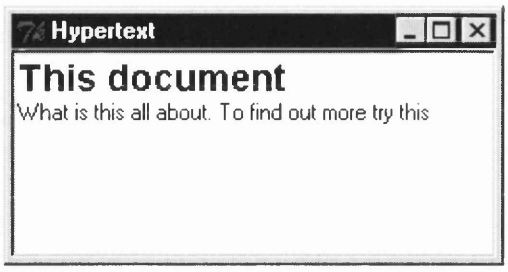

Figure 32 - A Hypertext Viewer in FranTk

In our simplified example, a Hypertext document consists of a list of hypertext tags. Hypertext entries are either headings, in bold, 14 point font; hyperlinks to an address; or simple text.

```
type Hypertext = [HypertextTag]
data HypertextTag = Heading String | Text String
                  | Link String Address
type Address = String
```

We can display a hypertext page as follows. We create a hypertext page by providing a hypertext behavior to display. When we find an address we pass the address to a listener, which will presumably change the page.

```
hypertextPage :: Behavior Hypertext ->Listener Address-> Component
hypertextPage hypertext readPage = do
 mkEdit [setIDocB $ lift1 todoc hypertext),readOnly True]
 where
  toDoc :: HyperText -> IDoc
  toDoc text = structured (SGroup (map (tag readPage) text))
```

We make an edit widget that displays the hypertext, using `setIDocB`. Recall that `setIDocB` displays an `IDoc` valued behavior. We convert the hypertext into structured text to display it. We make the hypertext page read only. We therefore need to convert each hypertext entry into a structured entry.

```
tag :: Listener Address -> HypertextTag -> Structured
```

Plain text is converted into plain structured text.

```
tag change (Text s) =  SText s
```

A heading translates into tagged text. We add an edit tag to the text, which sets the font *for that bit of text* to 14 point, bold.

```
tag change (Heading s) =
  STextTagged (s ++ "\n")
              (mkEditTag [font (namedFont "Helvetica" 14 [Bold])])
```

A hyperlink also translates into tagged text. In this case we make the tagged text blue, and bind mouse presses with button 1 to tell the change listener about the address.

```
tag change (Link s addr) = STextTagged s
                                (mousePress 1 (tellL addr change) $
                                 mkEditTag [foreground S.blue])

mousePress :: Has_Input w => Int -> Listener () -> w -> w
```

Finally we can make an instance of the hypertext editor. We make a BVar that holds the current page. Pressing a hyperlink will therefore cause the page to be set, by applying the `getURL` function which looks up the hypertext page associated with a given address.

```
hypertextviewer :: Component
hypertextviewer = do
          pagestate <- mkBVar init
          hypertextPage (behavior pagestate)
                        (mapIOL getURL $ input pagestate)

getURL :: Address -> IO Hypertext
```

As an `IO` action the `getURL` function could communicate without the outside world, perhaps through the Foreign Function Interface that allows Haskell to talk to other languages. The use of structured text makes it easy to define quite complex interfaces, such as hypertext browsers, in a simple declarative style.

4.13.5. Edit Marks - Referring to a moving location

An `EditMark` is a mark that can be placed in an edit area. It moves around as the text moves, and so is a way of marking important points in the text such as the beginning of a section. We can create an `EditMark` using `mkEditMark`.

```
mkEditMark :: [Conf EditMark] -> GUI EditMark
```

As with `EditTags` we can give an edit mark a unique identifier. We can set and access its location via the selection interface.

```
useMrkIdent :: MrkIdent -> Conf EditMark

instance HasSetSelection EditMark (Maybe TIndex)
instance HasGetSelection EditMark (Maybe TIndex)
```

Marks are useful for monitoring moving locations. They exist in many toolkits, and text editors such as the Gnu Emacs editor. For instance, we could create a mark at the beginning of a chapter in a document, and then monitor that position in a declarative manner using a FranTk BVar.

4.13.6. Interrogating a Document

FranTk provides a number of declarative operators to interrogate a document. For instance, given two behavior indices, we can extract a String behavior defining the text between those two indices, for all time. As a second example, we can find all the tags that cover a given index, *for all time*. Using behaviors we can therefore define a simple declarative interface to access a document[14].

```
getText :: DocumentB -> Behavior (TIndex,TIndex)-> Behavior String
getTagsAt :: DocumentB -> Behavior TIndex -> Behavior [TgIdent]
```

4.13.7. Putting it all together - A Structured Text Editor

FranTk has been used to develop a simple declarative implementation of a structure editor for a simple imperative language (see Figure 33).[15] Bernard Sufrin and Oege De Moor have developed a simple, purely functional, model of a structured editor [193]. This model was developed as an executable formal specification. They model the editor as a Tree, with two basic types of operations: navigation operations that alter which `Subtree` is selected; and update operations which modify the current subtree. For the purposes of our discussion, we need to know that all edit operations are of type Edit, the complete editor is of type `EStatus`, and the current subtree is of type `SubTree`. Finally, we need to know that there is a function `applyEdit`, which applies an `Edit` update to an `EStatus` to yield a new `EStatus` and a `SubTree` update.

```
applyEdit :: Edit -> EStatus -> (EStatus,SubTree)
```

[14] More will be said of the practicality of this interface in Section 8.1.4
[15] This is work jointly carried out with Oege De Moor at Oxford.

There has been a reasonable amount of work attempting to provide formal models, often executable, of interfaces such as text editors (eg [69]). This work has not, however, involved producing actual Graphical User Interfaces because available toolkits were too low level. We have developed a FranTk implementation of this editor, which maintains the declarative model. This demonstrates that FranTk is able to work at a higher level of abstraction that other toolkits. The prototype has inspired others to look at this area [106].

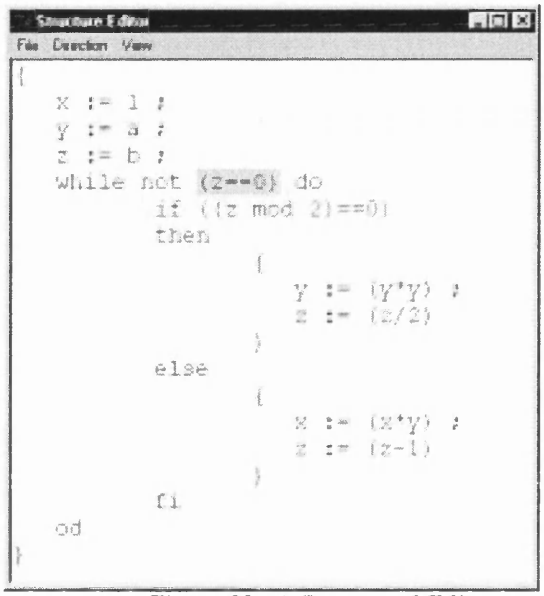

Figure 33 - A Structured Editor

The editor shows the currently selected subtree in grey. We can navigate around by typing, or by clicking on an area of text. For instance, clicking on "While" selects all of the "While do" loop.

To create an editor, we take an initial tree status and create a `Component`. To do this we first create a BVar; this will model the current status of the tree and the latest (`SubTree`) update. We convert this BVar into a Document Behavior using `fromEditTree`. We create a read-only edit widget, that displays the document behavior. We add a keyboard listener to this component that converts the keyboard input to Edit updates and then applies them to the tree status via the tree BVar's update-input listener.

```
mkEditor :: EStatus -> Component
mkEditor initTree = do
 treeBv <- mkBVar (initTree,emptySubTree)
 let docB = fromEditTree treeBv initTree
 keyPressAny (convertToEdits $
                comapL (applyEdit . fst) $ updInput treeBv))
   $ mkEdit [readOnly True,setDocB docB]
```

To convert key inputs to Edit updates, we accumulate a current command String by parsing it; this generates the update '`Just Edit`'[16] when a valid command has been fully entered. We then extract all of these updates using `mapMaybeL` (which performs map then filter).

```
parse :: (String,Maybe Edit) -> Key -> (String,Maybe Edit)
mapMaybeL :: (a -> Maybe b) -> Listener b -> Listener a

convertToEdits :: Listener Edit -> Listener Key
convertToEdits editL = toCommand $ mapMaybeL snd $ editL

toCommand :: Listener (String,Maybe Edit) -> Listener Key
toCommand l = scanlL parse ("",Nothing) $ l
```

[16] The value `Just` is part of Haskell's `Maybe` type, representing values that may or may not occur: `data Maybe a = Just a | Nothing`.

To display the edit tree we must convert it into a Document Behavior, using the `Structured` type.

```
fromEditTree :: BVar (EStatus,SubTree) -> EStatus -> DocumentB
fromEditTree treeBv initTree =
  let mkTg :: (EStatus,SubTree) -> Maybe (TIndex,TIndex)
      mkTg (eSt,_) = let x = getSelectedTag eSt
                     in Just (tindexTagFirst x,tindexTagLast x)

      select :: Listener Edit
      select = comapL (applyEdit . fst) $ updInput treeBv
  in
  mkDocumentB
   (structured ([STag (mkEditTag[backgroundB (grey 0.8),
                                 setSelectionB(lift1 mkTg bh)]),
                 viewTree select (subtreeOfEdit initTree)]))
   (event treeBv ==> \(_,t) ->
     replaceStructured (viewTree select t) (getTag t))
```

This definition has two important parts. We create an edit tag with colour grey to show the current selection. We set its indices, so that it covers the currently selected subtree. Each subtree has a unique name defined by its location. We refer to the start and end of the tag that covers the subtree, by using this unique name.

We can map a subtree to a `Structured` text view using `viewTree`. We also pass in a listener that expects to hear about any navigation updates on the tree. We need these as we wish to navigate to a subtree when we hear mouse clicks on one of the subtree keywords, such as "While". We implement these keywords using tagged text, and the rest in terms of normal text.

```
viewTree :: Listener Edit1 -> SubTree -> Structured
```

The structured text will be grouped with the subtree's unique name. To replace a subtree, we therefore replace the old subtree with the new one, using the location name.

As this example shows, the use of document behaviors and structured text provides a programming tool with great expressive power. It has allowed us to develop a relatively simple, very high level implementation of a structured editor. Though high level, the implementation is still efficient. It runs at a usable speed, even when only run with "Hugs", a Haskell interpreter.

4.14. Introducing true Concurrency

Most of the time the declarative concurrency provided by FranTk through behaviors and events is enough. There are, however, times when real pre-emptive concurrency can be helpful. FranTk provides support for using Haskell threads along with the declarative behavior and event model.

Consider the following example. We have an interface to a theorem proving tool. This includes a window with a text entry area to develop the proof and a button to run the prover. When we press the prove button we don't want the whole interface to hang. Instead it would be helpful to have the proof computation occur in a different thread allowing the user interface to continue reacting to input.

We can model this by having the main GUI thread fork off a worker thread to perform the computation. The worker thread needs to be able to return its value and update the relevant BVar within the user interface code by firing a listener. The GUI thread can't block waiting for a result from the worker thread and the worker thread shouldn't directly update the listener itself. If this were to happen we'd have to be worry about synchronisation issues within the GUI thread; otherwise the interface might hang waiting for data from some background process.

Instead we provide primitives to allow worker threads to communicate through channel variables with the GUI thread.

```
addCVarListener :: CVar a -> Listener a -> GUI (GUI ())
addChanListener :: Chan a -> Listener a -> GUI (GUI ())
```

These allow the listener to wait for input to appear on the channel variables. The `CVar` version can be used when the worker thread is only to return one value, as with our example above; the `Chan` version can be used if the worker thread is to return a whole stream of values. Values appearing in a `CVar` or `Chan` will be merged with the streams of values that occur from widgets such as buttons, guaranteeing that the simple semantics of the remaining FranTk code are maintained. In particular, this means that after BVars are updated we can be sure that changes to any behaviors will be propagated to the interface widgets before any further updates are made.

Note that it is only safe to have one thread, the GUI thread running FranTk GUI code. Other threads should not directly attempt to alter the interface, or the BVars and wires making up the interface model. Instead other threads talk to the GUI thread through the simple interface described above, updating the behavioral model of the interface data. This restriction is similar to the treatment of the Swing GUI thread and worker threads in Java. As with Java, actions within the GUI thread should be quick to perform. Heavy weight computation should instead be delegated to worker threads.

Note that the GUI thread can communicate with its worker threads by non-blocking means, such as sending requests down a channel.

In our example, we would therefore have code that looked something like the following. Imagine we have a function `runProof` that takes some data and generates a result, but is a heavyweight computation.

```
runProof :: ProofVal -> IO ProofResult
```

The `proveComponent` is the button that runs the proof. It takes a Behavior modelling the current proof and a listener that the proof result should be sent to, when the proof is complete. We make a listener that produces a worker. Every time it hears a proof value it creates a new `CVar` and then forks a worker thread to perform the calculation. This ends by telling its result to the `CVar`. We then add the proof result listener to the `CVar`.

```
proveComponent :: Behavior ProofVal -> Listener ProofResult
               -> Component
proveComponent proofB proofResult =
    mkButton [text "Prove"] (listen mkWorker)
  where
    listen :: Listener ProofVal -> Listener ()
    listen worker = snapshotL_ proofB worker

    mkWorker :: Listener ProofVal
    mkWorker = mkGUIL $ \ pval -> do
      cvar <- liftIO newCVar
      liftIO $ forkIO $ do res <- runProof pval
                           putCVar cvar res
      addCVarListener cvar proofResult
      return ()

newCVar :: IO (CVar a)
putCVar :: CVar a -> a -> IO ()
```

We can therefore introduce real concurrency into an application where necessary, without making the rest of the application unnecessarily imperative.

4.15. Alternative Design Choices

4.15.1. Replacing Listeners

The use of listeners in FranTk is an important design choice. They allow us to define a simple Component type, by passing in a consumer argument when creating a component. However, programming with listeners is less declarative, and less intuitive, than programming with Events.

Because listeners are consumers of event, not producers, functions such as `comapL` have initially strange type signatures.

An alternative would have been to introduce typed Components. Here a Component is an action that produces a WidgetB *and a semantic value.*

```
type Component a = GUI (WidgetB,a)
```

We therefore have the following definitions for some basic components.

```
mkButton :: [Conf Button] -> a -> Component (Event a)
mkScale :: [Conf Scale] -> Component (Event Int)
mkLabel :: [Conf Label] -> Component ()
```

When we create a button, we also get an Event generating an occurrence on every click. This definition uses the common trick of providing the value that the button will produce as an argument. In general, where a component would take a Listener as an argument, it now produces an Event as a result. However, when a Component does not produce any output, it has no associated Event and instead is of type `()`.

Geometric combinators must now not only compose displays; they must also compose their semantic output. The above and beside combinators therefore take an extra function argument that composes two semantic objects to form a new one.

```
above,beside :: (a -> b -> c)
             -> Component a -> Component b -> Component c
```

Another useful combinator is `mapC`, which applies a function to a Component's semantic value.

```
mapC :: (a -> b) -> Component a -> Component b
```

We can now define the example "Counter" from Section 4.4.
The button based "Counter" takes an Integer behavior to display, and generates a Component that produces Integer updates. When composing the semantic output we merge the events produced by the buttons, and ignore the useless value produced by the label.

```
composite :: Behavior Int -> Component (Event (Int -> Int))
composite b =
  let lbl = mkLabel [textB (lift1 show b)]
      incb = mkButton [text "inc"] increment
      decb = mkButton [text "dec"] decrement
  in (above snd lbl (beside (.|.) incb decb))
```

The scale component has the same interface as the composite counter above. It generates a new update, by mapping the `const` function across the component.

```
scale :: Behavior Int -> Component (Event (Int -> Int))
scale b = mapC const $ mkScale [scaleValB b]

const a b = a
```

Finally, we need to produce the counter, with the appropriate wiring. We merge the semantic events from both components , and form a behavior by stepping through the merged event, accumulating a value by applying the update function.

```
counter :: Component ()
counter = mapC (const ()) $ do
  fixGUI $ \ ~(_,e) ->
   let b = stepAccum 0 e
   above (.|.) (scale b) (composite b)
```

The operator that makes it all work is the `fixGUI` function. This is a GUI version of the Haskell `fixIO` function. It relies on laziness to allow its return value to be passed as an argument. We therefore need to guarantee that we only use the result of the function lazily.

```
fixGUI :: (a -> GUI a) -> GUI a
```

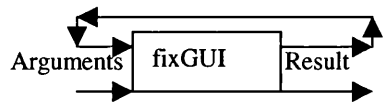

Figure 34 - The `fixGUI` Function

For small examples this approach appears more declarative. We can dispense with the need for listeners, adding only the minor inconvenience of adding `fixGUI`. We must simply be careful to only use the result of `fixGUI` lazily. This example can be simplified further with the introduction of recursive monadic bindings[47], which allow a set of mutually recursive monadic definitions. With this mechanism we could simply write the "`counter`" code as follows.

```
counter :: Component ()
counter = mapC (const ()) $ mdo
  let b = stepAccum 0 (event w)
  w <- above (.|.) (scale b) (composite b)
  return w
```

However, the type of our geometric composition operators is more restrictive. It becomes difficult to compose collections of components, particularly dynamic collections. We can define the following composition function. It merges all of its semantic events to form one composite event.

```
nabove :: ListB (Component (Event a)) -> Component (Event a)
```

However, for this to work we must tag and construct output into composite data types, and then untag it later on. This can easily become both unwieldy and inefficient. In addition, it is particularly unclear what to do about components with listeners in their configuration options. For instance, we cannot easily define an equivalent of `snapSelection`.

```
snapSelection :: HasSetSelection w i => Event a -> Listener (a,i)
              -> Conf w
```

More significantly, there is also no simple equivalent to the use of listeners in structured text `EditTags` (Section 4.13.4).

Though this design choice is a significant one, it is reversible. We can easily implement the alternative style discussed in this section on top of FranTk. For instance, we define `mkButton` as follows.

```
mkButton cs a = do
 w <- mkWire
 b <- FranTk.mkButton cs (input w)
 return (b,event w -=> a)
```

The Event style is perhaps better when handling small examples where all input components return the same sort of value. For instance, we could imagine doing this with a calculator where every button returned a calculator update function. In contrast, the Listener style scales better where it is more difficult to merge all input into a single type. Where helpful, therefore, we can easily combine both styles of programming.

4.15.2. Unifying Components and Widgets

We could do away with the need for a separation between widgets and components, by using the encapsulate function shown below.

```
encapsulate :: GUI WidgetB -> WidgetB
```

This is similar to the encapsulate function in Pidgets, which turns a value of type `St Widget` into a value of type `Widget`. This definition is permissible because a `WidgetB` is an abstract type that *when rendered*, will generate an interface. The `GUI` action will therefore be run, when the `WidgetB` is rendered. Two individual instances of the same widget will therefore have separate local states.

For instance, in the example below, `mkW` defines a counter button. It needs to be a `GUI` action because it produces some local state, the value of the counter. We can encapsulate the widget and display two of them side by side. As the local state is only produced, *when the composite widget is rendered*, both widgets will have their own local state, and will therefore maintain different counter values.

```
mkButton :: [Conf Button] -> Listener () -> WidgetB

example =
 let mkW :: GUI WidgetB
     mkW = do bv <- mkBVar 0
              return (mkButton [textB (lift1 show b)]
                               (tellL increment (updInput bv)))

     w = encapsulate mkW
 in beside w w
```

This approach is potentially useful because it allows us to unify two separate concepts. However, because of the prevalence of GUI actions in most FranTk applications, it is unclear whether it provides any real benefits in practice.

4.16. Conclusions

This Chapter has presented FranTk, a toolkit for developing graphical user interfaces in Haskell. It concentrates on providing a programming model that is both "declarative in the large and in the small" . Chapter 1 argued that declarative languages should attempt to handle both. Programming in a functional style is "declarative in the small"; structuring interfaces in terms of a set of declarative constraints is "declarative in the large".

Though FranTk uses the GUI monad and listeners which introduce imperative features, programming is still largely declarative. FranTk introduces the concept of a listener as an abstract value. Listeners allow imperative actions to be handled, but composed in terms of a functional algebra. The state of an application can be defined as a behavior value. These values can be easily composed. Unusually this extends to the ability to handle dynamic collections of objects as values, treating them in a functional manner. FranTk defines an interface in terms of components. These are constructed by passing in configuration options, including dynamic options. This allows us to define a component's appearance for all time. A Listener argument is also passed in when creating a component, thereby separating the semantic wiring from the visual Component. These components can therefore be geometrically composed using simple, pure functions. However, a Component represents an action that produces a widget. This allows it to have its own internal state.

FranTk therefore allows a compositional, declarative style of programming with both static and dynamic user interfaces.

Chapter 5 – FranTk Development Tools

This chapter presents two tools that can be used with FranTk to aid development. The first is a system architecture editor. This architecture is based on that provided by Clock and the tool is based on the ClockWorks development tool [134]. The second tool is a graphical widget builder, that allows static interfaces to be constructed visually, and then allows FranTk code to be automatically generated for the widget. Both tools were developed as proof of concept prototypes to attempt to demonstrate that visual development methods could be used with FranTk. They were developed to try to demonstrate the usefulness of the general concepts and so were not intended for use by other developers.

5.1. The System Architecture

The system architecture tool allows systems to be constructed as a tree of interaction objects. As with Clock an interaction object represents some widget on the screen, and the tree represents a hierarchical decomposition of the interface. A screenshot from the editor is shown in Figure 35. It represents a simple program that will display a button and a set of labels. The visibility of the whole interface is affected by `visRoot`. Each individual button or label will be visible when the root is visible and it itself is visible.

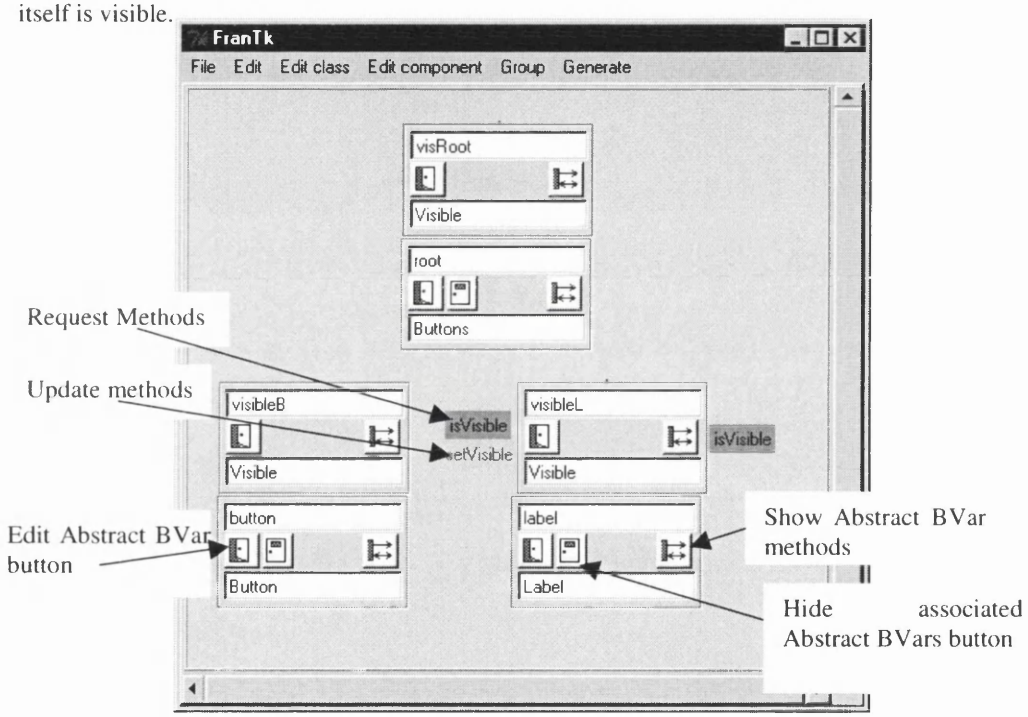

Figure 35 - The FranTk System Architecture Editor

Associated with each interaction object is a set of *Abstract Behavior Variables*. These maintain the state of the application. They represent the shared state available to the component and it's children. This makes it easy to decide what information should be local to an individual user, and what should be shared between several users.

Each Abstract Behavior Variable (Abstract BVar or ABVar) has a number of methods associated with it. These methods can be update or request methods. Update methods change the state of the object. Requests access the state of the object. Abstract BVars can be formed by composing together simpler abstract BVars. This provides good modularity and so supports component reuse.

In terms of Haskell, an Abstract BVar represents an abstract data type. The state of an Abstract BVar will be maintained internally using a set of FranTk BVars. Update methods are listeners that alter the state of the ABVar. Request methods are functions that may return Behaviors, Events or other values (such as other Abstract BVars).

To make use of the architecture tool, Haskell code is annotated to define which functions represent ABVar updates, requests and the actual ABVar data types. Each ABVar should have also have one constructor function to create values of the ABVar type. This constructor may take as arguments values that are request and update methods from other ABVars. The constructor function is annotated with the method names. These may also be specialised when individual instances of the ABVar are created. Unlike Clock, for individual instances these annotations may either mention just the method name, or may also mention the name of the ABVar providing the method. In this way, one Abstract BVar can use the value of another, and we can have more than one instance of an ABVar in scope at a time. Each component has a view function defining how it will be displayed.

As with ClockWorks the architecture tool takes care of distributing requests and updates appropriately down the tree. It traverses down the tree, matching component methods with their constructor uses. At each level it maintains a map of method names to ABVars to calculate which names are in scope. The tool then generates a control file that creates a component for each object in the tree, passing down the appropriate request and update methods as parameters.

As a brief example consider the architecture shown in Figure 35. It makes three separate uses of the *Visible* Abstract BVar. The interface for the Visible type is shown below. Each definition has a special, annotation comment associated with it. There are four types of annotation: (1) the *ABV* annotation, specifying the name of the ABVar and datatype; (2) a *Request* or (3) an *Update* annotation, which specifies the name of the method. Note that each method should take as its first argument the appropriate ABV; finally, (4) there is a *Cons* annotation that marks the constructor for the ABV. This takes the names of appropriate default methods that parameters will be linked to.

```
{- #ABV Visible -}
data Visible
{- #Request isVisible -}
isVisible :: Visible -> Behavior Bool
{- #Update setVisible -}
setVisible :: Visible -> Listener Bool
{- #Cons mkVisible isVisible -}
mkVisible :: Behavior Bool -> GUI Visible
```

When instantiating a value of type `Visible`, we can override the parameters in one of two ways: (1) replace the default method with another; (2) use a specific value in place of a method. There are two very common reasons when we may wish to override methods with a specific value. Firstly, there may be no suitable method in scope. For instance, in our example the `visRoot` ABVar makes an `isVisible` request, but as it is at the root there will be nothing to service the request. Secondly, when displaying a collection of children, we may wish to instantiate each with a separate value. In this case, the method would be overridden with a specific parameter. In our example, we will assume that the Label component is generated using this approach. This provides a reasonable level of flexibility. In contrast, Clock does not support the second style, and so a cumbersome workaround must be used.

Having defined the Visible type, we must now define each interaction object in the system. We define an interaction object in terms of two parts:

1. A general, reusable interaction object (annotated with `IObj`). This consists of a data type composing a set of abstract behavior variables. There is also a view function that defines how to display the interaction object in terms of this data type and its children. Note that the field name associated with each ABVar matches the name shown in the graphical editor. The data type and type definition of the view function will be automatically generated by the editor; the view function itself must be defined by the programmer;
2. Specific instantiations or *uses* of the object (annotated with `UseObj`). This will consist of two parts. The first of these is a type definition, defining how to construct a component for the interaction object. Parent view functions will be passed a function of this type to create children. This is important because each child instance may either receive an individual parameter value, or all may receive the same value. The latter can be automatically handled by the editor's code generator. Secondly, there is a constructor function to create the set of behavior variables for the component; the editor generates a best guess for this.

As an example, the *Buttons*, *Button* and *Label* interaction object definitions are shown below.

```
{- #IObj Buttons -}
data Buttons = Buttons {visRoot :: Visible}
viewButtons :: Viewbutton -> Viewlabel -> Buttons -> WComponent

{- #IObj Button -}
data Button = Button {visibleB :: Visible}
viewButton :: Button -> Component

{- #IObj Label -}
data Label = Label {visibleL :: Visible}
viewLabel :: Label -> Component
```

Recall that the `Visible` ABVar constructor makes one default request (`isVisible`). When instantiating the *root* interaction object, there will be no sources for the Visible ABVar to make its request. We must therefore instantiate it with a specific value. In this case, we choose to do this inside *mkroot*; we therefore pass no parameters in to the *mkroot* function.

```
{- #UseObj root -}
type Viewroot = WComponent
mkroot :: GUI Buttons
```

When instantiating a button, we do not override the default request method (`isVisible` is available from the parent). The `mkbutton` function therefore accepts a value of type `Behavior Bool` (the request value). The editor automatically generates a best guess for the definition of this function.

```
{- #UseObj button -}
type Viewbutton = Component
mkbutton :: Behavior Bool -> GUI Button
mkbutton bh mk = do {v <- mkVisible bh; return (Button v)}
```

The label object is more interesting. Each label instantiation is passed a separate Boolean behavior. The Viewlabel constructor function must therefore include this value as a parameter. Again the editor can generate a best guess for the definition of the `mklabel` function.

```
{- #UseObj label -}
type Viewlabel = Behavior Bool -> Component
mklabel :: Behavior Bool -> GUI Label
mklabel bh mk = do {v <- mkVisible bh; return (Label v)}
```

Finally, the editor will generate three functions that link together the tree of components. Note that `button` is passed its parameter by the `root` function; in contrast, `label` receives its from `viewButtons`.

```
root = do
  bs <- mkroot
  viewButtons (button (visRoot bs)) label bs

button vis = do
  b <- mkbutton vis
  viewButton b

label vis = do
  b <- mklabel vis
  viewLabel l
```

The annotations shown above will be automatically generated by the editor, when architectures are constructed using it. In contrast, if annotations are added textually to a file, the editor can reload the file and incorporate them. This provides a high degree of flexibility as a mixture of approaches can be used when building a system. The editor retains the advantages of Clock, as a system architecture can be built and viewed visually. In addition, it overcomes some of the restrictions in Clock, by providing support for composite abstract behaviors, fully parameterised interaction objects and explicit event routing.

We carried out no usability tests with the tool as it was a very early prototype with a number of cumbersome interface features. These problems would have served to mask the real issue of whether the visual programming style was useful. The tool was, however, used in the development of the case studies. It was overly cumbersome when dealing with small examples. However, when developing the ATC case study there did appear to be a need for it. Further work is, however, necessary to demonstrate the real usefulness of such a tool and technique.

5.2. The Interface Construction Tool

The second development tool that was prototyped was a graphical interface builder, which allows static interface components to be constructed visually. FranTk code can then be generated by the tool to integrate these components into an application. A screen shot from the tool is shown in Figure 36.

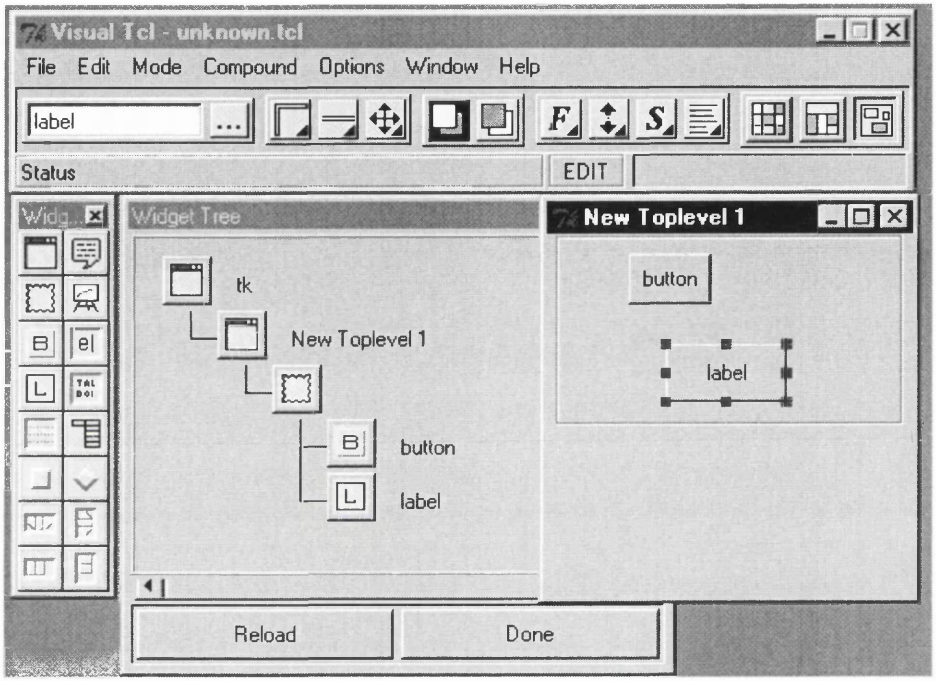

Figure 36 - The FranTk Interface Construction Tool

Rather than construct a new tool from scratch, an existing Tcl-Tk editor, Visual Tcl [7], was modified to provide the required functionality.

An interface component is first constructed visually, using the editor. Individual components can be provided with explicit names. Once the static component has been constructed, it is selected by clicking on the root frame of the item in the Widget Tree. The system then generates FranTk code for the component. In particular, it generates a constructor function which expects to receive configuration information for each explicitly named component in the composite component. The value returned by the constructor function depends on the object selected; for instance, if a top-level window were to be selected the constructor would generate a value of type WComponent. The editor generates a primitive Tcl-Tk constructor and explicit handles to access individual elements. These can be used to build and update the component.

From the programmer's point of view, the approach is fairly simple. For instance, in the example above, we have a button and a label on a frame background. If the button were named *mybutton*, the label were named *mylabel*, and the composite component were named *MyComponent*, then the editor would generate code with the following interface.

```
mkMyComponent :: ConfMyComponent -> Component
data ConfMyComponent = ConfMyComponent {
          mylabel :: [Conf Label],
```

```
mybutton :: [Conf Button]
}
```

To allow access to user input, we must extend the configuration options available to an object. For instance, we add an `onClick` option for buttons that adds a listener to the button's input. We provide similar options to access generic user input, and to listen to input on other components such as sliders.

```
onClick :: Listener () -> Conf Button
```

The interface construction tool therefore provides a simple but powerful tool for building complex static interfaces. The generation of such interfaces is still easier using a good visual tool, than by using a textual programming language. Unfortunately, again this tool had a number of basic usability problems which made it impractical to test it with other programmers. However, given a solid understanding of its quirks it did prove useful. Most of the basic interface components in the ATC system were developed with it.

The visual tools presented in this chapter are both proof-of-concept prototypes. They were developed to demonstrate that tool support could easily be provided for building FranTk based programmes, without any major restructuring of the language. In particular, the inclusion of an architecture editor demonstrates that a structured Clock-style architecture can be constructed on top of an existing embedded language, rather than requiring the creation of such a language from scratch. The inclusion of the GUI builder tool demonstrates that the use of a high-level language such as FranTk does not preclude the use of the very popular "Visual Interface Construction" approach.

Chapter 6 – Evaluating FranTk with The Case Studies

This chapter will present and discuss the three major case studies carried out with FranTk. It will then go on to evaluate FranTk in the context of the requirements outlined in Section 3.15. Finally, it will make some general remarks about how other FranTk users have found the system.

6.1. The Space Game in Fran

The Space Fighter Game provides the first case study covered in this thesis. This represented a test of support for building reactive, time based interfaces. The FranTk API subsumes most of Fran, as it provides support for both graphics and widgets[17]. We can therefore implement the game in a similar manner to Fran. (The Fran implementation was discussed in Section 3.13.2.) There are two important differences in the FranTk implementation:

1. We model all the lasers and enemies as dynamic collections. We use the dynamic Bag abstraction. Game objects (enemies/lasers/ship) have no implicit notion of equality. We could have arbitrarily added identifiers to each item; however, this would have been clumsy. The Bag abstraction provides an alternative solution. Recall that each object has a predicate specifying when it should be deleted, defined in terms of collisions with other objects on screen. We can use this predicate with the Bag abstraction to arrange the appropriate deletion of objects.
2. We replace image behaviors with canvas components

The FranTk implementation is therefore again simple and high level[18].

6.2. The QOC Editor

The QOC editor can be implemented fairly simply in FranTk. Recall that it allows several users to build a QOC rationale. Each user has a separate view of the QOC collection. Each QOC is maintained in a window. Different users can have windows open in different areas of the screen. Within each window, however, users' views are strictly WYSIWIS (What-You-See-Is-What-I-See). Users can also note their actions, and any extra textual information using a shared log. They can filter their own view, using the view menu. They can filter their view of nodes to show only Questions and Decisions; Questions, Decisions and Criteria; Questions and all Options; Questions, all Options and Criteria.

The level of sharing is important. Users can see changes made by anyone as they are being made, *in their current window only.* In contrast, they will only see the results of changes in the other visible windows. They can also see where the other users' cursors appear in their own window. Users have a colour associated with them, used for their cursor. A different colour scheme is used to represent changing objects. Objects being edited by a user appear in green; objects being edited by another user appear in red. Locking is at node level, so that two users can both act in the same window, but cannot both act on the same node in the window simultaneously.

We provide an abstract QOC model that each user will share. A QOC editor has a current named file, and a list of QOCs.

```
data QOCFile =
 QOCFile {filename :: BVar String, qocs :: ListBVar QOC}
```

[17] FranTk does not currently support 3-D images and Fran's sound interface. Given a sufficiently powerful underlying toolkit, there is no reason why it cannot in the future.

[18] Unfortuntely Tcl/Tk does not provide particularly good support for rapidly updated animations, so the FranTk implementation runs less smoothly than the game produced with Fran. This problem could be overcome by providing a FranTk binding to a toolkit with a more efficient graphics library, or by providing a more efficient Tcl/Tk graphics plug-in.

Each QOC has a unique, automatically generated identifier, a name, a list of nodes and links, and some attached notes (in the form of a dynamic document). It may be locked by a particular user, and so has some locking information associated with it. Recall that an object will be locked when being modified, and that partial changes will only be visible to users working within the same window. We therefore associate a value with the lock, that represents the current value of the modified object. In the case of a whole QOC, a user can change the name of the QOC. We therefore store a String with the lock. Finally, it will also have a current set of users, working on that QOC.

```
data QOC =
 QOC {unique :: Ident, name :: BVar String,
      nodes :: ListBVar Node, links :: ListBVar Link,
      notes :: DocumentBVar, lock :: LockBVar String,
      currentUsers :: SetBVar User}
type LockBVar a = BVar (Maybe (User,a))
```

Each Node has a name, node type (saying whether it is a question, option or criteria), a position (inside the QOC window) and some attached notes. Again it also has a unique identifier and a lock status. When locked a user may be modifying either the node position, its name or the node type.

```
data Node =
 Node {unique :: Ident, name :: BVar String,
       position :: BVar Point2, nodetype :: BVar NodeType,
       notes :: DocumentBVar,
       lock :: LockBVar (Either3 Point2 String NodeType)}
data NodeType = Q | O | C | D
```

Each Link connects two nodes, and has a link-type and label. Again it has a unique identifier and lock status. When locked, the user may be modifying the link type or name.

```
data Link =  Link {unique :: Ident, node1 :: Node, node2 :: Node,
                   linkType :: BVar LinkType, label :: BVar String,
                   lock :: LockBVar (Either LinkType String)}
data LinkType = Positive | Negative
```

We must also model the set of active users. We can do this using a dynamic set. Each user will have a name, a display (i.e. the name of the remote machine that they are working on), a cursor location (as each user can see the others' cursors) and a colour. Recall that each user has a colour for their shared cursor on other displays.

```
type Users = SetBVar User
data User = User {name :: String, display :: DisplayName,
                  colour :: Color, cursorLoc :: Behavior Point2}
```

Using FranTk, we can therefore provide a simple, declarative model of the QOC editor's shared state, and set of users. To create the interface, we provide each user with a view on to this shared state.

We define at each level how to display a single item (such as a UserView, QOC, Node or Link). We can then simply use multiple copies of this view at the level above. For instance, we define a function userView to display a single editor instance for one user. We can then define the set of user views as a pile of window components, with one for each user.

```
editors :: QOCFile -> Users -> WComponent
editors qfile users = pile (fmap (userView qfile users)
                                 (collection users))
userView :: QOCFile -> Users -> User -> WComponent
```

We use this approach at each level, filtering where appropriate. For instance, within each QOC a user can filter the set of visible nodes (e.g. to view on Questions). We do this by simply filtering the node and link list based on the node type, according to a behavior based predicate, which represents the current view mode. We do this using the filterB function.

```
visibleNodes :: Behavior (NodeType -> Bool) -> ListB Node
             -> ListB Node
visibleNodes pred nodes = filterB (behavior . nodeType) pred nodes
```

At each level, the view definition is fairly simple. The QOCs are defined as a set of windows. Each QOC will itself contain a canvas. This canvas will contain a set of nodes and links constructed as canvas components. Recall that extra comments about QOCs, nodes and edges can be added to shared logs. We must there create a subwindow, containing a multi-line edit widget for each notes document.

The definition of a component may require the creation of some local state. For instance, when viewing a QOC window we must define a behavior variable to model the current node filtering predicate. This BVar can then be passed to the filter menu and used to filter the visible nodes.

Each user's view has a current QOC window. This is modelled as a BVar with the unique name of the QOC. When displaying a component (QOC, node or link), we compare the component's QOC and the current QOC. If they are the same, we attempt to use the current value from the lock behavior variable, otherwise we use the normal value. Clearly, we can only use the lock value if the item is actually locked, otherwise we must still use the normal value.

```
getValue :: Behavior Ident -> Ident -> LockBVar a -> Behavior a
         -> Behavior a
getValue currqoc unique lockbv valueb =
  ifB (lift1 (== unique) currqoc)
      (maybeB valueb sndB (behavior lockbv)) valueb
```

As an example, we could now get the title of a given QOC using the `getValue` function.

```
qoctitle :: Behavior Ident -> QOC -> Behavior String
qoctitle currqoc (QOC {name = namebv,unique = unique,
                       lock = lockbv}) =
  getValue currqoc unique lockbv (behavior namebv)
```

We can implement node and edge views fairly simply. For instance, we can allow nodes to be moved by the user by applying the `draggable` function. This function is passed a BVar which will model the object's position, and a BVar to model whether the object is moving. When the left mouse button is pressed it starts the object moving; when the button is released it stops it moving. When the mouse is moved, if the object is in motion, it updates the position BVar. As it can be applied to any canvas component we therefore have a simple, reusable implementation of drag & drop.

```
draggable :: BVar Point2 -> BVar Bool
          -> CComponent -> CComponent
draggable posBv movingBv c =
 let startMoving = tellL True (input movingBv)
     stopMoving = tellL False (input movingBv)
     moveItem = input posBv `whenL` (behavior movingBv)
 in
 mousePress 1 startMoving $
  mouseRelease 1 stopMoving $
  mouseMove moveItem $
  moveTo (bvarBehavior posBv) $
   c
```

We can therefore provide a relatively simple implementation of a powerful graphical editor. The use of dynamic collections was particularly important in this case study. They enable us to provide simple declarative models of complex, dynamic systems. We were able to develop the code in a consistent style, providing a model of the shared abstract state of the system. The complete system was then composed out of the individual element views.

6.3. The ATC System

6.3.1. The Prototype Design

The Air Traffic Control system provided a much more significant test of FranTk. The ATC Prototype that we produced allows several controllers to work together. It supports up to two controllers, planning and tactical, in two adjacent sectors. This prototype was developed with the help of a Human Factors expert at the UK's National Air Traffic Service, who acted as a customer and provided a realistic set of requirements.

Recall that the interface provides a radar map of the sector with aircraft positions shown as *blips* (see Figure 3 in Chapter 2). These show the current location of an aircraft and its last three positions, giving a good idea of aircraft acceleration. Associated with each blip is a label known as a *datablock* showing the Aircraft Callsign, next sector (or last sector if the aircraft hasn't entered this sector yet) and current flight level. Datablocks can appear in different colours depending on the status of an aircraft. For instance, an aircraft under the control of a sector appears in black, an aircraft co-ordinating entry to a sector appears in blue. These labels will also show data-link error messages, and will highlight values undergoing data-link co-ordination.

By moving the mouse over a datablock, the controller can cause a *selected flight label* to appear. This shows more flight details, including downlinked flight parameters, and allows the controller to send flight clearance and co-ordination instructions. This direct provision of flight information reduces the need for radio communications between controllers and pilots. This allows controllers to keep their attention on the radar rather than being forced to move to the edge of the screen.

A more detailed *flight data plan* window is also available. It allows controllers to interact with the selected aircraft in a similar manner. It also shows more details, including downlinked controller preferences and the flight route. The plan shows information on the currently *hooked* (currently selected) aircraft, which will also be highlighted on the radar screen and on the Aircraft Display window.

Controllers can send data-link messages in a number of ways. They can send individual clearance messages. These can be either immediate (i.e. change now) or conditional (e.g. change by a given time). Controllers can also send composite messages using a *tactical data entry widget.*

The simulator maintains a model of all the aircraft in the system, and a separate model for each adjacent, computer controlled sector. Aircraft and adjacent sectors may behave as expected, or may mutate and misreact to messages sent by a controller, according to a prespecified set of probabilities. This is used to simulate failures during testing.

The case study therefore required the implementation of a large system, with a complex and powerful functionality.

6.3.2. The ATC Architecture

In developing the ATC system, we made use of all the FranTk development tools. A screenshot from the system architecture editor can be seen in Figure 37. It shows the architecture for the ATC system, structured as a tree with each box representing an interaction component. Recall that the structure represents the hierarchical display of the interface. The *Simulator*, consists of a set of *Controller Views.* (A single *Controller View* was shown in Figure 3.) This in turn consists of a *Radar Area*, showing all *Aircraft;* a *Messages In* Window and a *Messages Out* Window, showing received and sent datalink messages; a *Flight Data Plan* window; *a Data Link Messages* window; a *Configure* window (allowing controllers to modify certain aspect of the appearance of their screen); and *an Aircraft Display* window.

In the system, all shared data is associated with the *Simulator* component, and data local to each controller is held at or below the *ControllerView* level. The Simulator component is therefore a complex component in its own right, as it maintains the complete model of all the aircraft and simulated sectors. Each of these interaction objects will have Abstract BVars associated with it. For instance, a controller view shows information to a particular controller (*ATCUser*); has a *GroundSystem* associated

with its sector; a currently selected flight (*CurrentFlight*); and information about which windows are currently visible (*WindowsVisible*). This design allows for good application/interface separation. We have a complete model of the behavior of the ATC system, without any reference to its appearance.

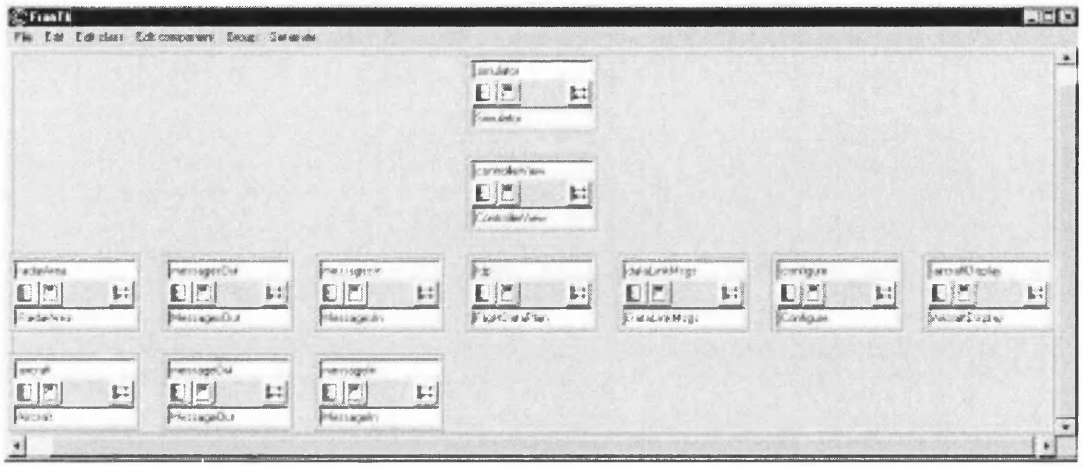

Figure 37 - The System Architecture for the ATC System

6.3.3. Building the ATC System in FranTk

Each user controls a sector. Their view is based on their sector's model of the aircraft. However, adjacent sectors and aircraft must react to communications. They will have behaviors and will generate messages. The simulation is therefore described as a collection of behaviors (representing aircraft and sectors) that communicate via events. Each adjacent sector and aircraft is modelled as a function which accepts messages via an event stream, and produces an event stream generating a set of response messages. Each aircraft maintains an abstract trajectory model, and generates downlink messages on its event stream to inform each interested sector of its flight parameters. The active aircraft are then modelled as a dynamic set. New aircraft are created on the basis of an alarm event, that goes off according to a plan (read in from an input file). Aircraft are deleted according to a predicate, which specifies when the aircraft leaves user controlled sectors and therefore ceases to be useful.

Each sector itself maintains a dynamic collection of aircraft which represents the sector's view of the airspace. New aircraft are added when a sector receives an abstract boundary message from an adjacent sector. New aircraft are deleted when they leave the visible airspace, again according to a predicate. The sector communicates with others by receiving messages on an event stream, and generating others on another event stream. We therefore have a consistent high level approach to modelling communication within the system; components are specified as functions from events to events.

A Sector will automatically generate certain messages. For instance, when a flight reaches a predefined distance from the next sector an advance boundary information message should be sent. This is specified as a predicate event.

```
abiMsg :: SectorPlan -> Behavior Location
       -> Behaviour FlightPlan -> Event ABIMsg
abiMsg sectorplan locB flightplanB =
  onceE (predicate (isWithinForABI sectorplan locB))
  `snapshot_` flightplanB ==> mkMsg
   where
         mkMsg fplan = ABIMsg fplan
```

This says that once only (*onceE*), when the location is within the sector plan, sample (*snapshot*) the current flight plan. Then generate an event, that is an *Abstract Boundary Information Message* (`ABIMsg`) containing the sampled flight plan.

A similar approach can be used to generate time outs. When a message is sent to an adjacent sector or aircraft, a timeout will be generated if the message has not been acknowledged within a given time. This again can be specified using a predicate; `predicate (time >* lift0 (t + timeout))`. This `time` value will have been provided by the `timeTick` function introduced in section 4.3.2. The existence of an explicit notion of time is therefore very powerful here. It allows us to program real time systems at a very high level of abstraction. This makes it easier to understand and reason about a prototype's behaviour, and so to find errors in a design.

The use of Fran behaviors proved very useful when developing the ATC system. Using behaviors, we can provide a simple, elegant model of an aircraft's trajectory. The aircraft model then simply snapshots the appropriate flight parameters when it needs to generate a flight-parameter downlink message.

For instance, we can model an aircraft's flight level as a composition of its cleared flight level and vertical rate of change. Given an initial flight level (`efl`), a cleared vertical rate of change (`cvrc`), and a cleared flight level (`cfl`), we can describe the actual flight level of the aircraft as shown below.

```
flightlevel :: Behavior Int
flightlevel = lift0 efl + roundB (atRate (fromIntegral actualvrc))

actualvrc :: Behavior Int
actualvrc = ifB (flightlevel <* cfl)
                cvrc
                (ifB (flightlevel >* cfl) (- cvrc) 0)

atRate :: Behavior Double -> Behavior Double
```

This says that the flight level changes at the rate of the `actualvrc`, and starts out at `efl`. The actual vertical rate of change (`actualvrc`) is equal to the cleared vertical rate of change (`cvrc`) if the flight is below the cleared flight level (`cfl`); it flies down at the cleared vrc (ie at - `cvrc`) if the flight is above the cleared flight level, and is 0 otherwise. These definitions can be mutually recursive (that is dependent on each other) because we are using a lazy functional language (where definitions are only evaluated when they are needed). Using behaviors we can therefore easily model and compose dynamic systems at a specification level, rather than at an implementation level.

A range of different component behaviours can be defined with FranTk. However, a common style is to define the behaviour of a component in terms of a state transition function, that maps input data, and the current state to a new state. These can be parameterised. For instance, *Clearance* models a flight clearance for a given parameter. A value has either been cleared, or is clearing. In the second case we have a clearing value, a copy of the old value, a clearing status, which says whether we're waiting for a logical acknowledgement message, or a pilot response, and the identifier for the message sent as a clearance. A specific instantiation of a clearance might be for a flight level clearance. This consists of an actual level, and a conditional value. This can be one of a range of values including `Now` meaning start climbing immediately, or `ByPoint`, meaning reach the specified level by the given point.

```
data Clearance a =
        Cleared  {clearedVal :: a}
      | Clearing {clearingInfo :: a,
                  clearingMsgId :: ! MsgId,
                  clearingStatus :: ! ClearanceStatus,
                  clearedVal :: a
                  }
data ClearanceStatus = WaitingLack | WaitingResponse | ...
type ClearanceFL = Clearance (Int, Cond)
data Cond = Now | ByPoint Location | ...
```

A clearance is altered by datalink messages (`DLMsg`). These can be clearances with a unique message identifier, and message body, or can be aircraft responses with a message reference. A pilot can agree to co-operate (`WILCO`) or refuse (`UNABLE`). A logical acknowledgement message will also be sent.

```
data DLMsg a = ClearMsg {msgId :: MsgId,
                         msgInfo :: a}
             | Response {msgResponse :: Response, msgRef :: MsgId}
             | ...
data Response = WILCO | UNABLE | LACK | ...
```

The behaviour of the Clearance is defined in terms of the state transition function, `handleClearance`. The definition below shows one transition. If we are clearing and waiting for a logical acknowledgement message, and we receive one, then we wait for a pilot response.

```
handleClearance :: DLMsg a -> Clearance a -> Clearance a
handleClearance (Clearing val msg WaitingLack oldval)
                (Response Lack ref) =
       Clearing val msg WaitingResponse oldval
```

The state is initially cleared, with a given value. Every time a `DLMsg` is received the `handleClearance` function is applied to the message and the current state and a state transition occurs.

```
state :: a -> Event (DLMsg a) -> Behaviour Clearance a
state val clearanceE =
   Cleared a `stepAccum` clearanceE ==> handleClearance
```

This modelling approach is very powerful. It allows the behavior of a system to be defined at a very high level of abstraction. The state-transition pattern is similar to the state machine model of many high-level specification languages [10]. It is important because, as shown later, it allows us to derive a formal specification.

6.3.4. Redesign

The ATC system was designed based on a set of requirements, discussed with a human factor's (HF) specialist, at the UK's National Air Traffic Services. I went down to NATS headquarters to perform some redesign with the HF specialist. This allowed us to investigate whether FranTk was capable of supporting any requested changes. It allowed us to test whether these changes could be carried out quickly enough to support rapid iterative design. Finally, it allowed us to test whether the quality of the resulting prototype would be sufficient for the needs of real users.

The results were very positive. I carried out a number of important changes to the prototype. I was able to make all the necessary requested changes; the HF expert was impressed by the speed with which changes were made, the interactive nature of the process and the quality of the resulting interface[19]. The most important aspect was the development of a tactical data entry widget, to allow rapid messages to be sent to a controller. The HF specialist had been considering such a widget for a while, but had not had access to the necessary resources to have it properly prototyped. We therefore spent some time working on the design, until he was satisfied. The resulting interface proved useful for him. He requested a copy of the final system, and as a result of this exercise the tactical data entry widget has been recommended as the primary method of tactical message composition in EOLIA (the NATS datalink project). The close user involvement therefore provided an important element of realism to the evaluation. We were able not only to demonstrate a successful application of FranTk, but to produce a prototype widget that has been successfully adopted by the end-user organisation.

The original data entry widget, and the new tactical data entry widget are shown in Figure 38.

The new widget allows the level, heading and speed to be set. The widget is opened to set a particular value, such as the level. When opened, the relevant field will be selected (Stage 1 in Figure 38). When the return key is pressed, if the value is invalid, the relevant field is highlighted in yellow (Stage 2 in Figure 38). If the value is valid, the widget enters "send mode"; all fields are shaded in grey, and the focus shifts to the send button (Stage 3 in Figure 38). The message may either generate a datalink

[19] A letter from the NATS HF Specialist discussing his experience is enclosed at the end of this Thesis.

message or simply a ground system update, depending on the value of the datalink checkbox. If one of the arrow buttons is pressed, a separate widget will appear to allow structured entry of the data. For instance, when the level button is pressed, the level widget will appear.

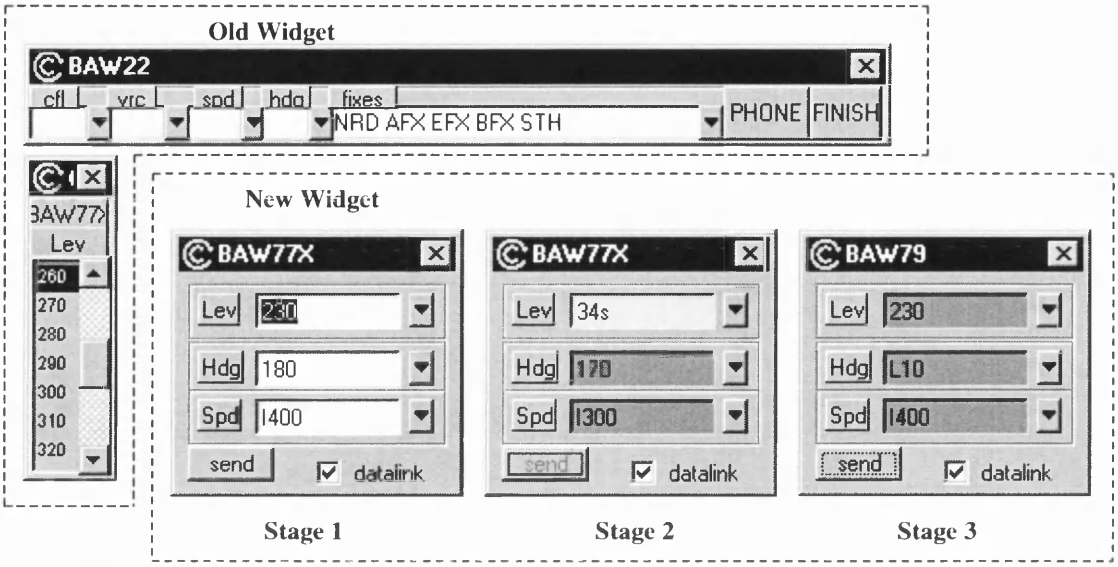

Figure 38 - Tactical Data Entry Widget

The existence of application/interface separation was very important to the successful integration of this component. While behaving differently, both widgets serve the same purpose. They can both be defined using almost the same type. We can therefore easily replace one with the other.

The original tactical data entry takes an event, which generates an occurrence, every time the widget is to popup, with a value specifying the current necessary defaults for the interface. The second parameter is a value of type `TDCall`. This provides a set of listeners which popup helper entry widgets, and a listener that should be told of the resulting data link message.

```
mkTDEntry :: Event RequestTactical -> TDCall -> Listener DLMsg
          -> WComponent

data RequestTactical = RequestTactical {
    tacticalFL ::  RequestInfo,
    ...}

data RequestInfo a = RequestInfo {
    requestCallsign :: CallSign,
    defaultVal :: a,
    ...}

data TDCall =
   TDCall {askLevel :: Listener (RequestInfo,Listener Level), ...}
```

The new tactical data entry widget was first created visually with the interface construction tool. This allowed it to be designed with the Human Factors specialist's direct involvement. The tool generated a constructor function, `mkTDE`, which creates a tactical data entry component.

```
mkTDE :: ConfTDE -> WComponent

data ConfTDE = ConfTDE {
    window :: [Conf Window],
    level :: [Conf Entry],
    levelB :: [Conf Button],
    ...}
```

The tactical data entry widget was then constructed using this interface. It requires one slight modification; when the widget is popped up, one of the fields will be initially selected. We therefore associate a value of type Open with the popup event, which specifies which field should be selected.

```
data Open = OpenLev | OpenHdg | OpenSpd

mkTDEntry :: Event (Open,RequestTactical) -> TDCall
          -> Listener DLMsg -> WComponent
```

The widget must maintain some internal state. It uses a BVar to record whether or not it is in "Edit Mode", i.e. whether a field is being edited; a BVar to store the current level, heading and speed values; a BVar to represent the value of the datalink checkbutton; and a BVar that represents whether the widget should be visible. Each of the current values is represented by a value of type "`Either a String`". This represents either the current value, or a String if the field contains an incorrect value.

```
data Either a b = Left a | Right b

mkTDEntry openE tdcall = do
  editmode <- mkBVarE True (openE -=> const True)

  level <- mkBVarE (Right "") (openE ==> const . extractLev)
  heading <- mkBVarE (Right "") (openE ==> const . extractHdg)
  speed <- mkBVarE (Right "") (openE ==> const . extractSpd)

  datalink <- mkBVarE True (openE -=> const True)
  visible <- mkBVarE False (openE -=> const True)
```

Each of the above BVars is updated by both its listeners, and by the open event. For instance, when the widget is initially opened, the visible BVar is set to `True`. We therefore use the `mkBVarE` constructor, which generates a BVar with an initial value that also depends on an initial event (initially presented in Section 4.6).

When the window is popped up, or if the level, heading or speed key is pressed when the widget is not in "Edit Mode", the focus will jump to the relevant entry field. We model the latter using a wire. Relevant keyboard input on the window will generate an occurrence on this wire when the widget is not in edit mode (the `whenL` combinator is similar to the Fran `whenE` combinator and is defined in a similar manner, in terms of `snapshotL` and `filterL`). We then simply merge the wire's event with occurrences from the open event.

```
keyInpW <- mkWire

let focusChangeE :: Event Open
    focusChangeE = openE ==> fst .|. event keyInpW

keyPress (mapMaybeL fromKey (input keyInpW)
            `whenL` notB (behavior editmode)) $ do
```

Here the function `fromKey` has the type `fromKey :: Key -> Maybe Open`.

The widget will only be in "Send Mode", when it is not in "Edit Mode" and all the fields contain valid entries.

```
let valid = lift1 isLeft (behavior speed) &&* lift1 ...
    isSendMode = valid &&* notB (behavior editmode)
```

We generate a Tactical Data Entry widget, with visibility dependent on the `visible` BVar. The visibility will be altered when the Tactical Data Entry window is closed, or when the "Send" button is pressed. Each field and field button has a similar behavior, defined in terms of `dispEntry` and `dispButton`. The buttons' behavior is fairly simple; it will fire the appropriate listener from the `TDCall` set, thereby opening the appropriate helper widget.

```
let confs = ConfTDE [title callsign,
                     onClose (tellL False (input visible))]
                    (dispEntry parseLev showLev editmode
                               level focusChangeE isLev)
                    (dispButton askLev reqInfo)
                    ...

ifB (behavior visible) (mkTDE confs) emptyComponent
```

An entry field is used to edit each of the three data values (level, speed and heading). Each has five important configuration options:

1. `textContents` - the entry will show the value of its BVar;
2. `backgroundColour` - when the field value is invalid the background will be yellow, otherwise if the widget is in edit mode the background will be white, otherwise the background will be grey;
3. `changeFocus` - when a field is opened, the keyboard focus will be set to that field;
4. `select` - when a field is opened all the text in that field will be selected;
5. `onReturn` - when the return key is pressed, the input will be parsed, and the relevant data value BVar will be updated. The widget will then leave edit mode.

```
dispEntry :: (String -> Either a String) -> (a -> String)
          -> BVar (Either a String) -> BVar Bool
          -> Event Open -> (Open -> Bool)
          -> [Conf Entry]
dispEntry parse show dataValBV editmode focusChangeE isRelevant =
  [textContents,backgroundColour,changeFocus,select,onReturn]
 where
  textContents =
        textB (lift1 (getValue show) (behavior dataValBV))

  backgroundColour =
        backgroundB (ifB isInvalid yellow
                         (ifB (behavior editmode)
                              white grey))
  changeFocus =
        setFocusE (filteredFocusE -=> True)

  select =
        setSelectionE (filteredFocusE -=> (Just (I 0,IEnd)))

  onReturn =
       snapReturn $
           mergeL (comapL parse $ input dataValBV)
                  (tellL False $ input editmode)

  filteredFocusE :: Event Open
  filteredFocusE = focusChangeE `suchThat` isRelevant

  getValue :: (a -> String) -> Either String a -> String
  getValue show (Left s) = s
  getValue show (Right n) = show n

  isInvalid = lift1 isRight (behavior dataValBV)
```

We therefore have a high level implementation of the tactical data widget. The application/interface separation in the implementation made it easy to incorporate a new tactical data entry widget, with only a little new coding. In general, the development of a large, complex case study was relatively easy in FranTk. We were able to construct the system using a consistent programming approach. The system was easily designed in terms of a set of components, which were integrated in a compositional manner. The use of dynamic collections to model the collection of aircraft, and datalink messages was again important. The system provides a number of different views of an aircraft's data and of the datalink message collection. For instance, the "Message In", "Message Out" and "Datalink Msgs" windows each show a separate filtered view of the sectors datalink message set. A separate view of each aircraft is

provided by the aircraft data block, selected flight label, Aircraft flight strip window, and the flight data plan window. The ability to provide multiple views of a dynamic collection was therefore very important. FranTk's support for real-time predicates proved particularly useful when developing the prototype. Predicates were used, for instance, to define time outs on datalink events.

6.4. Summary of Evaluation

This Chapter has shown how FranTk was able to cope with a set of case studies of increasing complexity. At each stage we discussed how FranTk coped with the programming issues that arose. We will now summarise how well it satisfies the requirements outlined in Chapter 3.

6.4.1. High level and declarative

FranTk provides a high-level approach to interactive systems design. The behavior of a system is defined in terms of a set of BVars. These can include dynamic collections, allowing a dynamic system to be modelled declaratively. The appearance of an interface is then defined as a function of this state. At no point do we require to imperatively define *how* an interface will change; instead, we simply state *what* it should look like. Imperative actions are handled using the listener type; listeners can be composed in terms of a high-level algebra of operators. The use of predicates is an important part of this declarative programming approach. We can provide a simple, high level specification of when a change should occur, rather than requiring to check and make changes using an imperative approach.

6.4.2. Declarative Concurrency

FranTk clearly supports declarative concurrency. The ATC system, for instance, consists of a number of concurrently evolving components (such as aircraft). These are modelled in terms of behaviors and events rather than requiring any explicit pre-emptive concurrency. FranTk provides support for explicit concurrency where required; however, the support for declarative concurrency was sufficiently powerful to support two different multi-user interfaces.

6.4.3. Compositional

FranTk provides a compositional programming style. Individual user interface components, are values of type `Component`. These may be composed using a set of declarative graphical combinators. However, they may also contain internal state. Interactive components are all defined in terms of functions that accept an abstract BVar (i.e. a collection of listeners, behaviors and events) and generate a value of type `Component`. We can attach listeners to the user input from a component, including a composite component; we can also apply a style (such as background colour) to a composite component. Components are represented by untyped values, we can therefore compose collections of components. While there is a separation between top-level windows, standard components, and canvas components, this is necessary because each of these represent a separate class of widget; it does not make sense to geometrically compose a top-level window and a button beside each other. We therefore have a very compositional programming style.

6.4.4. Component based application/interface separation

FranTk provides good application/interface separation. Application code in terms of BVars, and interfaces are defined as views of these BVars. The existence of dynamic collections is again very important here. In each of our case studies we were able to provide an abstract model of the system, and then provide multiple views of this data. This was particularly important in the ATC system, where there are many different views of the aircraft and datalink messages.

6.4.5. Visual Tool support

We presented two visual tools to aid the construction of interactive systems in FranTk; an architecture editor and an interface construction tool. They were implemented only as proof-of-concept prototypes. Both tools therefore had a number of low-level usability problems. They proved unnecessary in the first two case studies; however, they did appear useful in the development of the ATC system. The interface construction tool was the more useful of the two. The ATC system contained many popup-windows; all

of these were created with the tool. Such tools are commonplace in most interface development systems and so the desire for such a tool is less controversial[139]. The usefulness of the architecture tool is more questionable. The FranTk architecture does improve on Clock's architecture by providing support for composite abstract behaviors variables, fully parameterised interaction objects and explicit event routing. While it did appear useful in enabling me to understand the architecture of the ATC system, it was certainly not indispensable. Further work would be necessary to perfect the tool and convincingly demonstrate its importance.

6.4.6. Scalability

The FranTk language proved sufficiently scalable to handle a range of large examples. The ATC system, for instance, was constructed in a consistent style. The programming approach did not need to be radically altered to handle such a large system. Instead it scaled gracefully to handle the new system.

6.4.7. Efficiency

FranTk proved sufficiently efficient to handle all of the case studies discussed in this thesis. The ATC system ran sufficiently quickly for the NATS Human Factors specialist to use it for his purposes. All of the examples in Chapter 4 run efficiently even under the hugs interpreter. This includes the structured program editor discussed in Section 4.13.7. Unfortunately, the space fighter game does not run smoothly under FranTk, because Tcl-Tk does not support sufficiently fast animation. However, this is not a fundamental problem with FranTk, but could be overcome by providing a FranTk binding to a language with more efficient graphics support. The equivalent program runs well enough under Fran.

6.4.8. Platform independence

The FranTk library was developed on top of Tcl-Tk to provide a platform independent widget set. FranTk will run anywhere that Tcl-Tk, and Haskell will run. This allows the development of an interface with native look and feel which can run unaltered on Windows, Unix and Macs.

6.5. Areas for Further Work

6.5.1. Debugging

Currently FranTk provides only very primitive support for debugging. The issue of debugging in a lazy functional language is an area of research in its own right. Because values are evaluated lazily, and much Haskell code consists of pure functions, we cannot simply provide a program trace by placing print statements within program code. To attempt to overcome this problem all current Haskell implementations come with the (non-standard) function `trace`.

```
trace :: String -> a -> a
```

It can be applied to a value and prints a message when the given value is evaluated. FranTk provides equivalents for events and behaviors which print a message when the given event has an occurrence or when the value of the behavior is sampled. This approach has, however, a number of know problems. The trace function can result in incomprehensible output, it tends to be invasive and it can change the strictness of the things it is observing[68].

There are a number of promising areas of research within the Haskell community that are developing significantly more usable and powerful debugging tools. For instance, Gill[68] presents two tools Observe and HOOD that allow the observation of intermediate data structures after a program's execution. The ART project at York University is currently developing a tracer/debugger for Haskell. This work is based on an earlier prototype that is distributed with one Haskell compiler[189]. It is not immediately clear how easily the internal machinery of the FranTk implementation could be hidden when using such a tool. One interesting area of future research would be to investigate how easily such tools could be used in conjunction with a high level toolkit such as FranTk.

6.5.2. Exceptions

Recent work has introduced a new Exception mechanism into Haskell[160]. This allows exceptions to be thrown by any Haskell function, and to be caught within the IO monad. The use of behaviors and events makes the use of such exceptions difficult. For instance, imagine that an exception was thrown by Behavior value in a configuration option. It is not immediately clear where we should catch such an exception. One solution might be to allow exception handlers to be passed along with behavior values to generate configuration options. Further work is required to find the best mechanism with which to integrate exception handling into a FranTk program.

6.5.3. Usability of FranTk

It is important to note that all of the case studies discussed in this thesis were developed solely by myself. There is therefore no evidence that others could have achieved similar results with FranTk. There are a number of other programmers using it to develop a range of pieces of software. All of these programmers have had a solid understanding of Haskell: recall that FranTk was developed for those familiar with functional programming.

FranTk has been used to produce a number of text editing interfaces. An Msc student at Oxford has extended the structured editor discussed in Section 4.13.7. Another PhD student has also used it to develop a simple interface to a theorem prover. FranTk is currently being used by another researcher to develop a midi sequencer that allows musical scores to be edited visually and then played. It has also been used on a number of other smaller projects by both undergraduate computing students and researchers.

Feedback on FranTk has been generally positive. Those programmers that I spoke to have found it reasonably easy to learn at least the basic concepts; one user described how they were happy that they could "knock up a simple interface easily within half an hour". Most of the confusion that has arisen has been a result of FranTk's listeners. They seem to add an initial steep learning curve before programmers become happy with using them. This initial feedback has, however, been very informal. As outlined in Chapter 1, *this thesis does not attempt to evaluate the usability of FranTk as a GUI programming language*. Further work would be required to do this properly.

Myers et al[139] discuss the notions of "Threshold" and "Ceiling" as ways of categorising user interface tools. The "threshold" is how difficult it is to learn to use the system; the "ceiling" is how much can be done using the system. They argue that "most successful current systems seem to be either low threshold and low ceiling or high threshold and high ceiling". FranTk is clearly a high "Ceiling" system as it has allowed a range of significant systems to be developed. However, it would also appear to be a high "Threshold" system. Myers et al argue that tools should allow a "Gentle Slope" where new concepts can be learned incrementally rather than creating "walls" in which a developer must stop and learn many new concepts and techniques. There clearly are such "walls" in the use of FranTk. Further work is required to determine where they are and to either simplify the conceptual model or provide better tool support to overcome them.

6.5.4. Conclusions

In conclusion, FranTk satisfies all of the requirements set out in Chapter 3, and proved sufficiently powerful to handle a range of large case studies. It does, however, seem to have a fairly high threshold when programmers are learning to use it. Further work is required to determine what aspects of FranTk make it difficult to learn and how to improve its usability for programmers. Further work is also required to successfully integrate tools such as debuggers and language extension such as Haskell's new Exception mechanism.

Part III. Implementation

Part III of this thesis discusses the implementation of FranTk. It is separated into two Chapters.

Chapter 7 discusses the implementation of the core Functional Reactive Programming (FRP) combinators. It discusses the semantics of FRP. It then presents three different novel and cunning implementations, which are more efficient (both in terms of time and space) than existing FRP implementations. Each of the first two implementations have their problems. The third requires further research to determine whether it will really work. Chapter 7 also presents a clever approach to implementing dynamic collections. This Chapter is therefore aimed at readers interested in the difficult issue of implementing Functional Reactive Programming efficiently.

Chapter 8 discusses the implementation of the FranTk GUI library. Though FranTk has been implemented on top of Tcl-Tk, it has been implemented in as toolkit independent manner as possible. This Chapter is therefore of interest to readers wishing to understand how the FranTk GUI library works and those wishing to port FranTk to an alternative GUI toolkit.

Chapter 7 – Implementing Functional Reactive Programming

There are two major aspects to the implementation of Functional Reactive Programming within FranTk.

The first aspect is the provision of an efficient implementation of the core Functional Reactive Programming combinators[20]. The FRP programming style is very succinct and expressive. This unfortunately comes at a price. It is difficult to provide an efficient, robust implementation that is entirely faithful to the formal semantics.

The development of an efficient FRP implementation is important not just for FranTk. There are a growing number of other application areas to which the FRP approach has been applied. These currently include robotics[155], multimedia[200] and animation[44]. Current work at Yale University is investigating its application to Vision systems. The search for more efficient FRP implementations is therefore important to all of these application areas.

Section 7.1 will first discuss the semantics of FRP. There are two competing semantic models; one by Elliott and Hudak [44] and one by Wan and Hudak [206]. Both have implementations using lazy functional streams. Section 7.2 will then present a new data-driven implementation that provides significantly greater performance. Section 7.2.8 will show how to improve on this data-driven implementation by using weak references to make it more space efficient. Unfortunately, this implementation is not entirely faithful to either of the FRP semantic models, and requires changes to the types of some of the basic operators (Section 7.2.5). Section 7.3 presents an alternative solution which is a hybrid between the Elliott & Hudak functional streams implementation and imperative data-driven evaluation. This hybrid is still an unfortunate compromise, as it is not entirely robust. Section 7.4 outlines a third data-driven implementation that appears to be faithful to the Wan and Hudak semantics. Further work is required to verify that this implementation does indeed completely satisfy the semantics.

The second aspect is the efficient implementation of incremental behavioral collections. This will be discussed in Section 7.5.

7.1. FRP Combinators – A Semantics

The provision of a complete semantics for Functional Reactive Programming is an ongoing area of research. There have been several attempts to tackle the issue. Elliott and Hudak[44] presented a denotational semantics for the operators on events and behaviors. Wan and Hudak [206] present an alternative formal semantics for FRP which differs from the original semantics in a few key aspects. Finally, Daniels[35] devoted a PhD thesis to providing a complete semantics for a language CONTROL, based on Fran.

Daniel's CONTROL language uses a number of constructs not available in Haskell, and so cannot be implemented as an embedded language. His semantics also has many similarities to that provided by Wan and Hudak. This Chapter will therefore concentrate on the first two FRP semantic models. Both have implementations based on lazy streams. The next two sections will briefly present the two semantic models, they will explain the basic principles behind their implementation, and will highlight the important differences between them.

7.1.1. A First Semantics

7.1.1.1. The Basic Semantic Model

The original Elliott and Hudak paper assumes an abstract domain of polymorphic behaviors and events. They define an interpretation of a-behaviors as a function from time to a-values, producing the value of a behavior `b` at time `t`.

```
at : Behavior a -> Time -> a
```

[20] The FRP implementation represents joint work with Conal Elliott and Simon Peyton Jones.

They define an interpretation on a-events as simply non-strict `Time x a` pairs, describing the time and information associated with an occurrence of the event.

```
occ : Event a -> Time x a
```

Elliott and Hudak then define the semantics of the event and behavior combinators. Here we will only discuss a subset of them; specifically the primitive time behavior, behavior-event reactivity, behavior lifting and primitive events. Readers wishing a fuller discussion of their semantics are directed to [44].

7.1.1.2. Two Initial Combinators – Time and Reactivity

The simplest primitive behavior is `time`; `at[[time]]` is just the identity function on `Time`.

```
time : Behaviour Time
at[[time]]t = t
```

The next significant combinator is `untilB`, which defines reactive behaviors. Specifically, the behavior 'b `untilB` e' exhibits b's behavior until e occurs, and then switches to the behavior associated with e.

```
untilB : Behaviour a -> Event (Behaviour a) -> Behaviour a
at[[b untilB e]]t = if t <= te then at[[b]]t else at[[b']]t
  where (te, b') = occ[[e]]
```

Note that the inequality here '$t <= t_e$' means that the behavior changes after the event occurrence, not on it. This allows the definition of self or mutually-reactive behaviors.

7.1.1.3. A Lazy Streams Implementation

At this stage it is useful to introduce the lazy streams implementation[21]. The simplest implementation of behaviors and events could mirror their semantic representations.

```
data Behavior a = Behavior (Time -> a)
data Event a = Event [(Time,a)]
```

We assume, for the moment, the existence of some function, `occ`, that checks for an event occurrence *before a given time*. To sample a reactive behavior, at time `t`, we first check whether the event `e` occurs before `t`. If so, we sample the new behavior, `b'`, that is part of the event occurrence, and if not we sample b.

```
occ :: Event a -> Time -> Maybe a

b `untilB` e = Behavior sample
  where sample t = case (e `occ` t) of Nothing -> b `at` t
                                       Just b' -> b' `at` t
```

Unfortunately, this representation has two fundamental problems. It allows nothing to be remembered from one sampling to another. This is very unfortunate, as we can clearly make incremental progress when sampling a reactive behavior. If we sample at time `t`, and the reactive behavior has not changed, then we know that if we check at a later time `t'`, we need only check the event for occurrences between time `t` and `t'`. The second major problem occurs when trying to define `occ`. The `untilB` function will often need to know that an event has no occurrence before a given time. Unfortunately, this may happen before we know the actual time of the first occurrence. This results in an obvious contradiction. Instead we need to represent an event as a stream of possible occurrences, and pad it with non-occurrences. The `occ` function can now simply search this list looking for a genuine occurrence before a given time.

```
data Event a = Event [PossOcc a]
type PossOcc a = (Time,Maybe a)
```

[21] This section is a summary of the lazy functional streams discussion provided in [46].

We can define a behavior as a function that maps time streams to value streams.

```
data Behavior a = Behavior ([Time] -> [a])

ats :: Behavior a -> [Time] -> [a]
Behavior f `ats` ts = f ts
```

The primitive behavior `time` is implemented as the identity.

```
time :: Behavior Time
time = Behavior (\ ts -> ts)
```

To implement *reactive behaviors* we use the `occs` function, which is a list version of the `occ` function introduced earlier. We need to scan through the list of event occurrences, enumerating behavior samples.

```
occs :: Event a -> [Time] -> [Maybe a]

b `untilB` e =Behavior (\ts -> loop ts (e `occs` ts) (b `ats` ts))
 where
  loop ts@(_:ts') (e:es) (b:bs) =
       case e of
           Nothing -> b : loop ts' es bs
           Just fb' -> fb' `ats` ts
```

We will now return to the remaining FRP combinators defined by Elliott and Hudak.

7.1.1.4. Lifting

Elliott and Hudak define a set of functions for "lifting" functions defined on static values to analogous functions defined on behaviors. This lifting is accomplished by a family of lifting operators, defined for each arity of function. These apply a function `f` to the values of a set of behavior arguments at time `t`.

```
lift_n : (a_1 .. a_n -> b) -> Behavior a_1 .. Behavior a_n -> Behavior b
at[[lift_n f a_1 ... a_n]]t = f (at[[a_1]]t) ... (at[[a_n]]t)
```

The lifting combinators are implemented using the more general '`$*`' combinator. This provides function application for behaviors. For instance, `lift1` is implemented as follows.

```
lift1 :: (a -> b) -> (Behavior a -> Behavior b)
lift1 f b = constantB f $* b
```

To implement the '`$*`' combinator we sample the function and argument behaviors using the time stream, and then zip together the two resulting lists, combining elements by applying the function to the argument, using (`$`).

```
($*) :: Behavior (a -> b) -> Behavior a -> Behavior b
fb $* xb = Behavior (\ts -> zipWith ($) (fb `at` ts) (xb `at` ts))

($) :: (a -> b) -> a -> b
f $ a = f a
```

7.1.1.5. Primitive Events

There are two basic primitive events: predicates and external events. Predicates are events that occur at the first time that a Boolean behavior becomes true, *after a given time.*

```
predicate : Behaviour Bool -> Time -> Event()
occ[[predicate b t_0]] = (inf {t > t_0 | at[[b]]t},())
```

External events are again functions of time. The meaning of an event `lbp` t_0, for example, is the pair (t_e, e) such that t_e is the time of the first left button press after t_0.

```
lbp: Time -> Event ()
```

It is important to note that in this model all primitive events have a start time. We must therefore pass this start time around explicitly. For instance, we would define a colour cycling function, which switches between red and green, as follows. Note that in order to catch successive mouse clicks we must pass the time of the last click to the next cycle.

```
cycle t0 c1 c2 =
  c1 `untilB` lbp t0 *=> cycle c2 c1

(*=>) :: Event a -> (Time -> a -> b) -> Event b
```

This quickly becomes cumbersome[22]. Later work on Fran modified the notion of primitive events such that they all depend on an environment (`User`) which provides both the input stream and the start time `t0`.

```
lbp : User -> Event ()
predicate :: Behavior Bool -> User -> Event ()
```

This environment argument must again be passed around as a parameter. It must be *aged explicitly* after every event. *Ageing* causes each input stream within the `User` argument to drop all occurrences before the time that the event occurred.

```
nextUser :: (User -> Event a) -> (User -> Event (a,User))
```

We can also explicitly age events as well.

```
withRestE :: Event a -> Event (a, Event a)
```

Elliott introduced a range of operators such as *switcher* that hide this ageing process from the user.

```
switcher b e = b `untilB` withRestE e ==> \(b,e) -> switcher b e
```

We will discuss the implementation of primitive events in Section 7.1.4.

7.1.2. A Second Semantics

While explicit ageing can frequently be hidden by such combinators, it can be a pain. Wan and Hudak [206] present an alternate semantics for FRP which introduces implicit ageing. They parameterise all behavior and event combinators with a start time and environment.

In their model, the meaning of a behavior, is a function mapping a start time, an environment (Elliott's `User` argument, which provides access to all input to the FRP system) and a time of interest to a value. Start times relate to the reactive nature of FRP. In '`b `untilB` e`', if an event occurrence `(t,b')` of `e` causes the overall behavior to switch to `b'`, we say that `b'` starts at time `t`. A behavior is unaware of any event occurrences that happened before its start time.

```
at : Behavior a -> Env -> Time -> Time -> a
```

The meaning of an event is a function that also takes a start time `T` and a time of interest `t`, and returns a finite list of time-ascending occurrences of the event in the interval `(T,t]`. An event occurring precisely at start time `T` is therefore not detected.

```
occ : Event a -> Env -> Time -> Time -> [Time x a]
```

To implement this model they too use a lazy streams implementation. The core data types, `Behavior` and `Event` are implemented as:

---

[22] It also results in an inherent space leak in an implementation, as we must hang on to an ever growing stream of user input values.

```
type Behavior a = User -> [Time] -> [a]
type Event a = User -> [Time] -> [Maybe a]
data User
```

Note that these definitions are similar to the earlier ones. The important difference is that the environment argument is now implicit rather than explicit.

To understand the changes that this model introduces it is sufficient to consider the new semantics of `untilB`.

The semantics of `untilB` is now as follows. The behavior '`b 'untilB' e`' exhibits b's behavior until `e` occurs, and then switches to the behavior associated with `e`. We have one significant difference, the new behavior will start afresh at the time of e's occurrence.

If occ[[e]] T t = $[(t_1,b_1),(t_2,b_2),..,(t_n,b_n))]$, then for any $r \in [T,t]$

```
at[[b untilB e]] env T r = at[[b]] env T r, n=0 or r <= t1
                           at[[b1]] env t1 r, otherwise
```

Reactivity can be implemented as follows.

```
untilB :: Behavior a -> Event (Behavior a) -> Behavior a
fb `untilB` fe =
  \user ts -> loop user ts (fe user ts) (fb user ts)
   where
    loop user ts@(_:ts') ~(e:es) (b:bs) =
      let user' = age user in
      b:case e of Nothing -> loop user' ts' es bs
                  Just fb' -> tail (fb' user' ts)

age :: User -> User
```

Note that the `user` argument is aged at each step, and that when the new behavior `fb'` is started it uses the aged `user`.

7.1.3. Comparing the Two Semantics

The difference between these two semantics can be seen clearly in the following example. Consider the behavior definition. The behavior `b`, switches from `0` to `b1` on a right button press. The behavior `b1` switches from `1` to `2` on a left button press.

```
b = let b1 = 1 `untilB` lbp -=> 2
    in 0 `untilB` rbp -=> b1
```

Consider the case when the user presses the left button, then the right button. What is the value of b after the right button has been pressed. There are two possibilities depending on which of the two semantics is used. If `b1` hears only input after the right button is pressed (i.e. it is restarted when the right button is pressed) then b will have the value `1`. However, if b1 remembers about earlier input then it will have the value `2`.

We can achieve either effect with either semantics using different definitions. To get the value `1` as the result, with Elliott and Hudak's semantics, we must explicitly age the `User` argument.

```
b u = let b1 u = 1 `untilB` lbp u -=> 2
      in 0 `untilB` withRestE (rbp u) ==> \(_,u') -> b1 u'
```

With Wan and Hudak's semantics, we get the value `1` by default.

```
b = let b1 = 1 `untilB` lbp -=> 2
    in 0 `untilB` rbp -=> b1
```

To get the value 2 with Elliott and Hudak's semantics we use the same `User` argument in the definition of `b` and `b1`.

```
b u = let b1 = 1 `untilB` lbp u -=> 2
      in 0 `untilB` rbp u -=> b1
```

To achieve the same effect with Wan and Hudak's semantics we must make use of the `runningIn` combinator. This takes a behavior and a function, using that behavior, and returns a new behavior. It starts the argument behavior and the result behavior at the same time.

```
runningIn :: Behavior a -> (Behavior a -> Behavior b) ->Behavior b
at [[runningIn b f]] env T t = at [[f (start b env T)]] env T t

at [[start b envStart TStart]] env T t =at [[b]] envStart TStart t
```

We can now define `b` as follows. In the example, `b1` is started at the same time as `b`, and so it hears input before the right button press as well as input afterwards.

```
b = let b1 = 1 `untilB` lbp -=> 2
    in runningIn b1 (\b1 -> 0 `untilB` rbp -=> b1)
```

The two semantic models therefore differ in their treatment of ageing: the Elliott-Hudak model uses *explicit* ageing of behaviors. The Wan-Hudak model uses *implicit* ageing.

7.1.4. Implementing Primitive Events

To create an event with an imperative handle, we use `newPrimEvent`. This uses a channel to allow communication. The event is initially created with one non-occurrence at time `t0`. To tell the event about future occurrences, we simply write values to the channel. To access all the occurrences of the event, we use `getChanContents`. This turns a channel into a lazy stream of values. We can access values from this stream, provided that we only ever evaluate the stream as far as the latest entry that has been written to it.

```
getChanContents :: Channel a -> IO [a]

newPrimEvent :: Time -> IO (PossOcc a -> IO (),Event a)
newPrimEvent t0 =
  do ch <- newChan
     -- The following entry is in case the event gets queried at
     -- time t0.
     writeChan ch (t0, Nothing)
     contents <- getChanContents ch
     return (writeChan ch,Event contents)
```

Unfortunately, using a lazy streams implementation results in poor performance, as it relies on a *demand-driven* approach. In order to actually run a program, based on this implementation, we must sample every behavior at every time interval. We must also pad every event with a non-occurrence once per sampling. This makes it difficult to define new primitive events, such as a FranTk *wire*. We must pad these new primitive events with non-occurrences as well.

7.2. Efficient FRP Combinators

7.2.1. Implementation Requirements

To allow an efficient implementation of the FRP combinators, we must move instead to a *data driven* approach. The Pidgets system, for instance, works in this way [177]. In the functional streams implementation, values are pulled from the behavior set. We must therefore sample every behavior at every time interval. In contrast, with a data-driven approach, changes are pushed towards constraint variables. When user input occurs it updates the values of behaviors and events. We therefore only need

to update areas of the screen that depend on behaviors and events which have actually changed. It also removes the need for padding of events with non-occurrences.

For FranTk, we need to support the following two imperative operators. Firstly, we need to be able to create a `Wire`. Secondly, we need to be able to add a listener to an event, so that its action will be performed on every event occurrence. Clearly, the second of these would be greatly aided by some form of data driven approach.

```
newWire :: IO (Wire a)
addListener :: Event a -> Listener a -> IO Remover
```

Section 7.2.2 will first introduce the Listener implementation. Section 7.2.3 will then go on to present a data-driven event implementation. Section 7.2.5 will discuss the problems with this implementation. This interface also allows us to implement efficient data driven behaviors. If we know what events a behavior depends on, we can invalidate and therefore evaluate it only when its dependent events occur. We will see how this can be done in section 7.2.7.

7.2.2. Implementing Listeners

A naïve implementation of listeners might follow their simple semantics as a callback, that performs an action with some event occurrence.

```
newtype Listener a = Listener (Occ a -> IO ())

type Occ a = (Time,a)
```

This implementation has a number of problems. Fundamentally, it is not sufficiently sophisticated to allow the implementation of all the Listener combinators discussed in Chapter 4. In particular, there are three important classes of combinator that we must implement.

The first of these classes includes any listener that should only perform an action a given number of times. This includes `onceL`.

```
onceL :: Listener a -> Listener a
```

The `onceL` combinator requires some mechanism to guarantee that its callback action is only performed once. This can be implemented by allowing a listener access to an *unsubscribe* action, that notifies the object talking to it, that it is no longer interested in hearing values.

```
newtype Listener a = Listener (Remover -> Listener' a)
type Listener' a = Occ a -> IO ()
```

Using such an unsubscribe action we can then implement `onceL` as follows. The `onceL` combinator creates a new listener. The first time this new listener hears a value, it fires its action and then unsubscribes from any future values. The `onceL` combinator therefore creates a listener that will perform its action once for every source it is added to.

```
onceL :: Listener a -> Listener a
onceL (Listener l) = Listener (\rm v -> do {l v;rm})
```

The second class of listener combinator includes `scanlL`. Recall that `scanlL` is a listener equivalent of the `scanl` function. The listener's current values starts with the initial value provided. Every time the listener consumes a value, it applies its update function to its current argument and the new value. It then uses the result to update its current value, and passes it to the argument listener.

```
scanlL :: (a -> b -> a) -> a -> Listener a -> Listener b
```

This form of listener therefore has some implicit state, a current value. We need to create this implicit state before the listener is applied for the first time. We can do this with the following listener representation

```
newtype Listener a = Listener (IO (Listener' a))
type Listener' a = Occ a -> IO ()
```

We can implement this as follows. First we create the callback action for the argument listener. We then create a new mutable variable (`IORef`), to hold the current state. Its initial value is `a`. Every time the new listener is fired, it reads the value from this variable (using `readIORef`), updates it (using `writeIORef`), and then fires its argument listener.

```
scanlL f a (Listener act) = Listener $ do
  op <- act
  ref <- newIORef a
  let tell (t,b) = do a <- readIORef ref
                      let a' = f a b
                      writeIORef ref a'
                      op (t,a')
  return tell
```

The third class involves a switching listener. Here it is important that a listener can both create some internal state, and return a remove action that should be performed by any enclosing listeners when unsubscribing. An efficient implementation of `switcherL` relies on knowledge about event termination, and so will be discussed in section 7.2.6.3.

Composing these three representations, we end up with the following listener representation. We can make "structural optimisations" by including a `NeverL` constructor. For instance, applying a function, such as `map`, to a `NeverL` listener returns a `NeverL` listener. We can therefore eliminate some unnecessary work. For ease of use we define a simple constructor function `mkPrimL`, and a simple destructor `getPrimListener`. We also define three extra combinators, `mkL`, `mkL'` and `getListener'`. The first of these creates a Listener from a simple callback function. The second and third are equivalents of `mkPrimL` and `getPrimListener`, except that they ignore the `Remover` return value. This is important because the remover return value is only used by `switcherL`, and is therefore otherwise just an unnecessary complication.

```
type Listener' a = Occ a -> IO ()

data Listener a = Listener (Remover ->IO (Remover,Listener' a))
                | NeverL

mkPrimL :: (Remover -> IO (Remover,Listener' a)) -> Listener a
mkPrimL f = Listener f

getPrimListener :: Listener a -> Remover->IO (Remover,Listener' a)
getPrimListener (Listener mk) rm = mk rm
getPrimListener NeverL rm = return (return (),const (return ()))

mkL' :: (Remover -> IO (Listener' a)) -> Listener a
mkL' f = mkPrimL $ \rm -> do {op <- f rm;return (return (),op)}

getListener' :: Listener a -> Remover -> IO (Listener' a)
getListener' l rm = fmap snd $ getPrimListener l rm

mkL :: (a -> IO ()) -> Listener a
mkL f = mkL' $ \ _ -> return (\ (_,a) -> f a)
```

We can now define most of the listener combinators in terms of a few primitive listener operations. These are `liftL1`, `liftL2`, and `liftLIO`. These compose listeners and allow us to apply functions over them. The first `liftL1` takes one listener and produces another by redefining its internal callback action.

```
liftL1 :: (Listener' a -> Listener' b) -> Listener a -> Listener b
liftL1 _ NeverL = NeverL
liftL1 f l = mkPrimL $ \rm -> do
 {(rm',op1) <- getPrimListener l rm;return (rm',f op1)}
```

This can best be seen by example. For instance, we can define `comapL` in terms of `liftL1`.

```
comapL :: (a -> b) -> Listener b -> Listener a
comapL f = liftL1 $ \act (t,val) -> act (t,f val)
```

The function `liftL2` allows us to merge two listeners and redefine their behavior.

```
liftL2 :: (Listener' a -> Listener' b -> Listener' c)
       -> Listener a -> Listener b -> Listener c
liftL2 f l1 l2 = mkPrimL $ \rm -> do
  {(rm1,op1) <- getPrimListener l1 rm;
   (rm2,op2) <- getPrimListener l2 rm;
   return (rm1 >> rm2,f op1 op2)}
```

We can define `mergeL` using `liftL2`.

```
mergeL :: Listener a -> Listener a -> Listener a
mergeL NeverL l = l
mergeL l NeverL = l
mergeL l1 l2 = liftL2 (\act1 act2 val -> do act1 val;act2 val)
                      l1 l2
```

The `liftLIO` function creates its callback action, and then passes that, along with a remover to its argument function, to generate a new callback action. Note that the remove action passed is the sequential composition of the top level remove function, and the remove action returned from `l`.

```
liftLIO :: (Remover -> Listener' a -> IO (Listener' b))
       -> Listener a -> Listener b
liftLIO _ NeverL = NeverL
liftLIO f l = mkPrimL $ \rm -> do
  (rm1,op) <- getPrimListener l rm
  op <- f (rm >> rm1) op
  return (rm1,op)
```

We can define the `onceL` and `scanlL` combinators in terms of it.

```
onceL = liftLIO $ \ rm op ->
 return $ \occ -> do op occ;rm

scanlL f a = liftLIO $ \_ act -> do
  ref <- newIORef a
  let tell (t,b) = do
       a <- readIORef ref
       let a' = f a b
       writeIORef ref a'
       act (t,a')
  return tell
```

If we need to fire a listener manually we can achieve this via `getListener`. This provides the listener via an IO action. It creates a mutable variable that will contain the listener's callback action. When we create the callback action, we pass in a remove action that sets the variable to contain a null action. To talk to the listener we read the contents of the variable, and apply the current action to the occurrence. Clearly this will do nothing when the listener has been removed.

```
getListener :: Listener a -> IO (Occ a -> IO ())
getListener NeverL = return (const (return ()))
getListener l = fmap snd $ fixIO $ \ ~(rm,_) -> do
 op <- getListener' l rm
 ref <- newIORef op
 let kill = writeIORef ref (const (return ()))
 let tell occ = do {act <- readIORef ref;act occ}
 return (kill,tell)
```

We therefore have a fairly simple yet powerful listener implementation. We still need a definition of `snapshotL` to snapshot behaviors, which will be provided when discussing demand driven behaviors in section 7.2.7. We can also extend this algebra with listener valued switchers. Such an implementation relies on the event termination operators, and so will be discussed in Section 7.2.6.

7.2.3. Data Driven Events

We now require to implement efficient, data driven events. The fundamental event operation that we must support is `addListener`. This suggests a simple, elegant representation. We make events subscription functions for listeners.[23] The implementation of `addListener` is therefore trivial. We can perform important structural optimisations if we extend the definition with an explicit constructor for `NeverE`.

```
data Event a = Event (Listener a -> IO Remover)
             | NeverE
type Remover = IO ()

addListener :: Event a -> Listener a -> IO Remover
addListener NeverE _ = return (return ())
addListener _ NeverL = return (return ())
addListener (Event add) l = add l
```

This implementation has a major advantage. We can use event listener duality to implement the event combinators. Recall from section 4.5.2 that when adding a listener to an event, we can either alter the event with an event combinator, *or the listener using a listener combinator.* We first define a simple primitive event combinator, which maps a listener combinator across an event. When we add a listener to the event, we first apply the listener combinator to the argument listener, before adding the resulting listener to the internal event (Figure 39). Clearly given a `NeverE` event we need do nothing.

```
mapLE :: (Listener b -> Listener a) -> Event a -> Event b
mapLE _ NeverE = NeverE
mapLE f e = Event $ \l -> addListener e (f l)
```

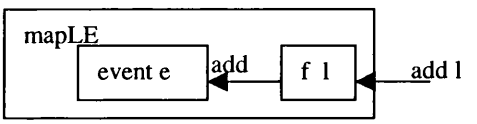

Figure 39 - The `mapLE` function

Using this approach we can, for instance, define `mapE` and `filterE` in terms of `mapLE` and the equivalent listener combinators.

```
mapE :: (a -> b) -> Event a -> Event b
mapE f = mapLE (comapL f)

filterE :: (a -> Bool) -> Event a -> Event a
filterE f = mapLE (filterL f)
```

This approach breaks down with `mergeE`, because though there is an equivalent listener operation (`mergeL`), there is no simple relationship between the two. We can, however, still provide a simple and elegant definition of merge on events. When adding a listener to two events, we add the listener to both and combine the remove actions. We can again optimise by using the knowledge that `NeverE` is a left and right identity.

```
mergeE :: Event a -> Event a -> Event a
mergeE NeverE e = e
mergeE e NeverE = e
mergeE (Event add1) (Event add2) = Event $ \l -> do
   rm1 <- add1 l;rm2 <- add2 l;return (rm1 >> rm2)
```

[23] This elegant representation was suggested by Simon Peyton Jones.

One other combinator that cannot simply be defined in terms of its listener equivalent is `switcherE`, the event level switcher.

```
switcherE :: Event a -> Event (Event a) -> Event a
```

Recall that the equivalent listener level switcher is of type

```
switcherL :: Listener a -> Event (Listener a) -> Listener a
```

We must instead define a new listener combinator, `switchL`. This takes an event and a listener. It produces a listener that consumes events. It starts by adding `l` to `e`. Every time the composite listener is told of a new event, it switches `l`'s interest to the new event, and drops its interest in the old one. Note that this passes its argument listener a remover that will delete it from the current event, rather than the remove action which will delete it from the switching event `ee`.

```
switchL :: Event a -> Listener a -> Listener (Event a)
switchL e l' = mkPrimL $ \rm -> do
 removerRef <- newIORef (return ())
 (rmn,op) <-getPrimListener l'(callRemover removerRef (return ()))
 let l = mkL' $ \_ -> return op
 rm <- addListener e l
 writeIORef removerRef rm
 let f (_,newEv) = do
       rmnew <- addListener newEv l
       callRemover removerRef rmnew
 return (rmn >> callRemover removerRef (return ()),f )

callRemover ref rmnew = do
              {rmold <- readIORef ref;writeIORef ref rmnew;rmold}
```

We can then define `switcherE` in terms of this primitive.

```
switcherE e NeverE = e
switcherE e ee = mapLE (switchL e) ee
```

This creates a new event, which whenever a listener is added begins initially listening to `e`. It also adds a separate listener to `ee`. Whenever this event-valued event occurs, it drops `l's` interest in the old event and adds `l` as a listener to the new one. We can again optimise using the knowledge that a switcher will not change on a `NeverE` event, and so will instead behave simple as `e`.

This representation of an event therefore says nothing of its meaning; it only defines how it can be used. In contrast, the functional streams representation contains only the event semantics, and says nothing about how it can actually be used. Events are given meaning when we create a `Wire`. Recall that a `Wire` is a primitive FranTk object, that contains two ends: an event end representing a stream of occurrences and a listener end that talks to the event.

```
data Wire a = Wire (Listener a) (Event a)
```

To create a wire we therefore need to give meaning to the subscription function. To do this we create a mutable set. The listener end tells every element in the set about new occurrences; the event end will add listeners to the set. Note that when a listener is added, we pass the unsubscribe action to the listener.

```
type ListenerSet a = MutSet (Occ a -> IO ())

newWire :: IO (Wire a)
newWire = do (set :: ListenerSet a) <- newMutSet
             let add :: Listener a -> IO Remover
                 add l = fmap snd $ fixIO $ \ ~(rm,_) -> do
                  (rmextra,fire) <- getPrimListener l rm
                  rm <- addToMutSet set fire
                  return (rm,rm >> rmextra)
```

```
        tell :: Listener a
        tell = mkL' $ \_ -> return $ \occ ->
                                foreachInMutSet set ($ occ)
    return (Wire tell (Event add))
```

This definition relies on the existence of some mutable set with the following interface. Adding an element returns a remove action. It is important that this remove action is idempotent, so it can be passed around numerous times. It is also important that actions are performed in first-in-first-out (FIFO) order. These operators also need to be efficient. Using a doubly-linked list representation we can implement these functions, with O (1) complexity for insert and delete, and O(n) complexity for traversal.

```
data MutSet a
addToMutSet :: MutSet a -> a -> IO Remover
foreachInMutSet :: MutSet a -> (a -> IO ()) -> IO ()
```

There are two further operators on mutable sets that we will need later. The first, `killMutSet`, clears the set and kills it so that no new items can be added to it. It is important that killing the set really clears it out, and invalidates every remove action, so that the actions no longer refer to, and therefore sustain items in the set. The second operation returns the current size of the mutable set; it will return `Nothing` if the set is dead.

```
killMutSet :: MutSet a -> IO ()
sizeMutSet :: MutSet a -> IO (Maybe Int)
```

7.2.4. A Problem with Efficiency

The above event representation is very elegant. However, it suffers from a serious efficiency problem. (It also has a semantic problem discussed in the next section.) We are applying combinators to listeners rather than to the event. This means that on every event occurrence, a combinator will be applied *once per listener, rather than once only.* This results in a mild performance problem with `mapE`, but in a very significant performance problem with `filterE`. A filter will be redundantly applied to every listener of an event (as shown in the left diagram of Figure 40). If an event has a large number of listeners, this will result in a lot of unnecessary work. We need to introduce some mechanism to guarantee that filters are only applied once per event occurrence (as shown in the diagram on the right of Figure 40.)

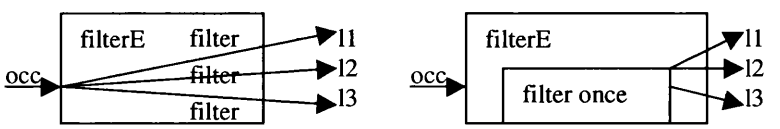

Figure 40 - Caching Events

We can solve this problem by introducing caching. We can define a simple cache action that creates a new wire, and makes the old event talk to the new one. Again we optimise when the event is a `NeverE` event.

```
cacheIO :: Event a -> IO (Event a)
cacheIO NeverE = return NeverE
cacheIO e = do
 w <- newWire
 addListener e (input w)
 return (event w)
```

This cache action is an IO action. However, our event combinators are all currently pure functions. We therefore need to overcome this by using the special Haskell function `unsafePerformIO`.

```
unsafePerformIO :: IO a -> a
```

This takes a Haskell IO action, and turns it into a simple value. The action will be performed when the value is evaluated. Clearly careless use of this function will break referential transparency, as the order

of evaluation now becomes important. Use of this function is therefore only usually safe, when we can guarantee that the argument IO action will produce the same result irrespective of when it is evaluated.

We can therefore define a simple `cache` function, and a cached version of `filterE`, as shown below. The next section will discuss where and when it is safe to use this `cache` function.

```
cache :: Event a -> Event a
cache e = unsafePerformIO (cacheIO e)

filterE f e = cache (mapLE (filterL f) e)
```

Unfortunately, this `cache` action, while improving time efficiency, can cause a space leak. Using caching we generate chains of events. Consider the following definition.

```
addListener (onceE (mapE f (filterE g e))) l
```

When simplified, this translates to the following code.

```
do w <- newWire
   addListener e (filterL g (input w))
   addListener (event w) (onceL (mapL f l))
```

It generates an event chain, as shown below. The event `e`, talks to the cache via filtered listeners; this cache in turn talks to the listener `l`. After the first occurrence, `l` will be removed from the cache. The cache is therefore no longer used. Unfortunately, the cache's listener will still remain in `e`, unnecessarily.

```
e -> filterL -> cache -> (onceL (mapL f l))
```

Similarly consider the following definition. This will result in an event chain. However, after the first occurrence of `e`, it will delete the cache's listener. However, `l` will still remain in the cache.

```
addListener (filterE f (onceE e)) l
```

Such event chains become very common with caching. They will continue to be built throughout the duration of a program, and if not also broken down, will result in a space leak. The second of these can be dealt with using the event termination mechanisms discussed in Section 7.2.6. The first of these can be dealt with using weak references discussed in Section 7.2.8.

We can also implement a cached version of `mapLE`, that caches its latest value without any need to create an entirely new listener set.

```
mapShareE :: (Listener b -> Listener a) -> Event a -> Event b
mapShareE _ NeverE = NeverE
mapShareE f e = unsafePerformIO $ do
 -- make the caching ref
 ref <- newIORef (error "nothing cached")

 -- now add a single listener that updates this ref, when
 -- the event fires, applying f. Note: that because listeners
 -- are handled in FIFO order this is safe. It's the first
 -- thing to get done with the cached event.
 let setcache ref a = writeIORef ref a
 addListener (mapLE e f) (mkL $ setcache ref)

 -- finally return an event, based on the old one that reads
 -- the value of the cache
 return $ mapLE e $ mapIOL (const $ readIORef ref)
```

7.2.5. A Problem with Laziness

7.2.5.1. Dealing with event history

As noted in the previous section, the data-driven event implementation applies any combinator for each occurrence, once per listener, rather than once only. Caching can be used to overcome this. This works well for combinators such as `mapE`, `mergeE` and `mapMaybeE`. These event combinators do not depend on an event's history. In other words, when processing an occurrence we do not need to know about the value, or even the existence of any other occurrence.

There are some combinators, however, that depend on an event's history and therefore on its start time. These include `onceE`, `scanlE` and `switcherE`.

As an example, consider the following definition of `scanlE`. It creates an event that accumulates a value, based on a function, an initial value and an event.

```
scanlE :: (a -> b -> a) -> a -> Event b -> Event a
scanlE f a e = mapLE (scanlL f a) e
```

Every time a listener is added to the event, it will begin to separately accumulate its own value. Two listeners added at different times would therefore consume different values. For instance, consider the following example.

```
testscanl = do
  w <- newWire
  op <- getListener (input w)
  let l1 = (mkL (\x -> print ("one",x)))
      l2 = (mkL (\x -> print ("two",x)))
  let e :: Event String
      e = scanlE (++) "" (event w)
  addListener e l1
  op (0,"a")
  addListener e l2
  op (1,"b")
```

This will produce the "Output 1" shown below. With the Elliott-Hudak FRP semantics, we would expect to see "Output 2" instead.

```
("one","a")
("one","ab")
("two","b")

  Output 1
```

```
("one","a")
("one","ab")
("two","ab")

  Output 2
```

Why? We add the first listener. It will create its own internal state, to hold the current accumulated value. We then fire `op`, causing the first listener to print "a". Now we add the second listener. This now creates its own internal state, to hold its accumulated value. When we fire `op` a second time, listener one will have accumulated a second value, producing "ab". However, listener two will only have heard the "b" value, as it will not know about the event history. That is, it will not know about the occurrences that have been produced before it was added. This combinator does not therefore satisfy the Elliott-Hudak FRP semantics.

In contrast, with the Wan-Hudak semantics an event must be provided with a start time. If we assume that `addListener` provides this start time then "Output 1" above conforms to their semantics.

7.2.5.2. Introducing Caching

When we add the caching mechanism from Section 7.2.4, we get an alternative problem. Consider a cached implementation of `scanlE`.

```
scanlE f a e = cache (mapLE (scanlL f a) e)
```

When evaluated this will create a cached event, adding a listener to `e` that accumulates a value. Listeners added to the scan event will now listen to the cached event. Each listener will therefore hear the same value. The `testscan1` example will now produce the output expected by the Elliott-Hudak semantics (Output 2).

However, we could easily imagine rewriting the `testscan` function as shown below. Here we have replaced each use of `e` with the full `scanlE` based definition. Now this would again produce Output 1 described above. These two definitions therefore produce different results. Unfortunately, in a referentially transparent language, these two definitions should be equivalent. In fact an optimising compiler might replace either definition with the other, during either an inlining process, or if it were to perform some form of common sub-expression elimination.

```
testscan3 = do
  w <- newWire
  op <- getListener (input w)
  addListener (scanlE (++) "" (event w))
              (mkL (\x -> print ("one",x)))
  op (0,"a")
  addListener (scanlE (++) "" (event w))
              (mkL (\x -> print ("two",x)))
  op (1,"b")
```

We therefore have a fundamental caching problem. The use of caching is only safe when used with combinators that do not rely on an event's history.

7.2.5.3. A Solution by Redefinition

The current representation therefore has a fundamental problem. It attempts to ignore the notion of "start time" entirely. This is a serious flaw. In the presence of caching the event combinators are not referentially transparent. We need an alternative solution to solve this problem.

One solution is to change the type of every history based combinator, to move it into the `IO` (or `GUI`) monad.

```
scanlE :: (a -> b -> a) -> a -> Event b -> IO (Event a)
scanlE f a e = cacheIO (mapLE (scanlL f a) e)

onceE :: Event a -> IO (Event a)
onceE e = cacheIO (mapLE onceL e)
```

These definitions are now safe. The start time of each new event is provided by the IO action. We can describe the semantics of these combinators in terms of either the Elliott-Hudak or Wan-Hudak semantics. Here we will use the Elliott-Hudak semantics. If we assume the existence of a definition of `scanlE` (such as the functional streams one) named `FRP.scanlE`, then we can define this new `scanlE` version in terms of it. First, we get the current time, `t`. We then use `afterTime`, which drops every event occurrence before `t`. We then make a scanning event based on this new event. This definition therefore explicitly removes all of the event history, before creating the new event.

```
scanlE f a e = do
 {t <- getTime;return (FRP.scanlE f a (afterTime e t))}

afterTime :: Event a -> Time -> Event a
```

This discussion brings to light an important issue in the definition of `addListener`. What exactly is its semantics? We really only wish to fire the listener on any future occurrence of event `e`. Conceptually, we drop the event's history before adding the listener. The definition of `addListener` therefore involves an implicit use of `afterTime`. Here `addListenerPrim` is assumed to be some primitive implementation of addListener that applies its listener argument to every event occurrence.

```
addListener e l = do {t <- getTime;
                      addListenerPrim(afterTime e t) l}
```

This also has an important consequence for event-listener duality. The equivalencies defined in Section 4.5.2 only hold if the given combinator does not use the event history, or does not have one. For instance, the `onceE` equivalence only holds, if we first drop all previous occurrence from the event. This happens because of the implicit `afterTime` in the definition of `addListener`.

```
do {t <- getTime;addListener (onceE (afterTime e t)) l}
== addListener e (onceL l)
```

By redefining the type of all history based FRP combinators, to move them into a monad, we can thereby overcome the semantic problems discussed in this section. These changes also have a knock on effect on the type of some behavior combinators, discussed in section 7.2.7. This redefinition changes the flavour of the language, by making `IO` actions more prevalent. However in FranTk, all widget code is defined in terms of the `GUI` monad. We can therefore easily integrate these `IO` based combinators into our code, at the expense of making the examples in the previous chapter a little more imperative in appearance. Section 7.3 will present a hybrid implementation that attempts to faithfully satisfy the Elliott-Hudak semantics. Section 7.4 presents a refinement of this data-driven implementation that attempts to faithfully satisfy the Wan-Hudak semantics.

7.2.6. Event Termination

On some occasions we can know when an event has terminated. In particular, this is important when "connected" with caching. Recall the following example from Section 7.2.4. It has been rewritten to take into account the new type of `onceE`.

```
do e <- onceE e
   addListener (filterE f e) l
```

This forms an event chain, with two cached events. However, after the first occurrence of `e`, the entire chain can be broken down as the `onceE` definition will terminate the event. We can use this to prevent the second of the space leaks, inherent in the use of caching.

7.2.6.1. Specifying Event termination

We can specify the rules for event termination in terms of the event algebra[24]. There are three important combinators that will affect event termination.

1. When applying `onceE` to an event, the new event will terminate when either `e` terminates or after the first occurrence of `e`.

   ```
   termE (onceE e) == e -=> () .|. termE e
   ```

2. When merging two events, `e1` and `e2`, the combined event will terminate when either `e1` terminates, and then `e2` terminates; or when `e2` terminates and then `e1` terminates. (Here the operator >> is the monadic event combinator, defining sequencing, discussed in Section 4.5.2.)

   ```
   termE (e1 .|. e2) ==
     (termE e1 >> termE e2) .|. (termE e2 >> termE e1)
   ```

3. When applying switcher to an event, the new event will terminate when `ee` has terminated, and the current (last) event has terminated. We define this by maintaining the current event in the event-valued behavior `currE`. When `ee` terminates, we snapshot it and then wait for the last event to terminate.

   ```
   termE (switcherE e ee) ==
     do {e <- termE ee `snapshot` currE;termE e}
    where currE :: Behavior (Event a)
          currE = e `stepper` ee
   ```

[24] This approach was inspired by a suggestion from Conal Elliott.

We therefore have an elegant specification of the rules for event termination. We simply add a termination event to a general event, and add a clean-up listener to the termination event, where necessary, to delete any data structures that are no longer necessary.

```
data Event a = NeverE | Event (Listener a -> IO Remover) TermE
```

We can give `TermE` the following interface. We can create a termination event. This creates a terminate event, and a listener that can be fired to terminate the event. We can add a listener to a terminate event that performs its action when the event terminates.

```
data TermE
mkTermE :: IO (Listener (),TermE)
onTerminate :: TermE -> Listener () -> IO Remover
```

For instance, we can then define a function `newPrimWire`. Given a termination event, it creates a new wire. This new definition, adds a listener to the terminate action, that clears the listener set of the event. This deletes all the event listeners, and kills the set so that no new listeners can be added. It then passes the termination event on with its new event.

```
primNewWire :: TermE -> IO (Wire a)
primNewWire termE = do
  lset <- newMutSet
  let add = ...
  let tell = ...
  onTerminate termE (mkL_ (killMutSet lset))
  return (Wire l (Event add termE))

killMutSet :: MutSet a -> IO ()
```

We can then define the standard `newWire` function in terms of this combinator. Here we pass in a termination event that will never terminate.

```
newWire :: IO (Wire a)
newWire = primNewWire neverTermE
neverTermE :: TermE
```

7.2.6.2. Implementing Event termination

We cannot simply implement a termination event as a standard event because we get caught up in a recursive definition. We can, however, use the same approach. If we are careful, we can include extra optimisations, as we know that the termination event will only ever fire once, when the event dies.

We can define a termination event as follows. It is useful to also have an IO action saying whether the event has now terminated.

```
data TermE = TermE {onTerminate_ :: Listener () -> IO Remover,
                    isTerminated :: IO Bool}

neverTermE :: TermE
neverTermE = TermE (\_ -> return (return ())) (return False)

onTerminate :: TermE -> Listener () -> IO Remover
onTerminate (TermE add _) l = add l
```

We create a termination event in a manner very reminiscent of the `newWire` implementation. It has three important differences. Firstly, the termination event has a boolean variable that records whether its event has already terminated. We therefore need to set this variable to `True` when we fire the termination action. Secondly, we can safely assume that an event will only terminate once. After we have run each termination listener, we can clear out the listener set. Thirdly, the termination listener may have been added to more than one event. However, we know that after it has been fired once, we do not need to wait for any of the other termination sources to fire. We can therefore safely remove the

termination listener from all its sources. Recall that primitive listeners are constructed with the following function.

```
mkL' :: Remover -> IO (Listener' a)
```

Every time we register the listener, we receive as an argument the remove action to unsubscribe the listener; we then generate the actual listener callback using this. In this case we want to store the unsubscribe action and then return the actual termination action. We also store the listener in a variable. Note that this termination action therefore performs 5 different actions: (1) it tells all its termination listeners about the event termination; (2) it sets the termination boolean variable to True; (3) it clears out the termination event's listener set; (4) it clears its own listener from all the termination sources that it is listening to; (5) finally, it sets its listener variable to contain a null listener. This means that if its listener is ever added to another source again, it will do nothing.

```
mkTermE :: IO (Listener (),TermE)
mkTermE = do
 var <- newIORef False
 mset <- newMutSet
 dieRef <- newIORef (return ())
 let add l = fixIO $ \rm -> do
       op <- getListener' l rm
       addToMutSet mset (op )

 tellref <- newIORef neverL

 let fire = mkL' $ \rm -> do
      b <- readIORef var
      updIORef dieRef (\oldrm -> oldrm >> rm)
      return $ \v -> do
       {foreachInMutSet mset ($ v);
        writeIORef var True;
        clearMutSet mset;
        callRemover dieRef (return ());
        writeIORef tellref neverL}

 writeIORef tellref fire

 let fireL = mkL' $ \rm -> do
        l <- readIORef tellref
        getListener' l rm

 return (fireL,TermE add (readIORef var))
```

We now redefine the three special event combinators that will affect event termination. The definition of event merging is the simplest. We merge the two events and merge their terminators.

```
mergeE :: Event a -> Event a -> Event a
mergeE NeverE e = e
mergeE e NeverE = e
mergeE e1 e2 = Event (merge e1 e2)
                     (mergeTerm (termE t1) (termE t2))
```

To merge two event terminators we must do two things. The merged termE will have terminated when both of its arguments have terminated. To add a termination listener to the merged termE we add a modified listener to both. The termination listener for one event will only be fired if the other has already terminated.

```
mergeTerm :: TermE -> TermE -> TermE
mergeTerm t1 t2 = TermE add terminated
  where
   terminated = do
    b1 <- isTerminated t1
    b2 <- isTerminated t2
    return $ b1 && b2
```

```
add l = do let mk :: IO Bool -> Listener () -> Listener ()
               mk isterm = liftL1 $ \op v ->do
                             b <- isterm
                             if b then op v
                                  else return ()
           rm1 <- onTerminate t1 (mk (isTerminated t2) l)
           rm2 <- onTerminate t2 (mk (isTerminated t1) l)
           return $ (rm1 >> rm2)
```

To understand the definition of `onceE` , we must recall that it now has an `IO` based definition. We now explicitly create a new wire that will hear the single event occurrence, and add new listeners to it. To redefine `onceE` we also create a new termination event. When the original event terminates, or has an occurrence, we tell this new termination event. We use this new termination event when creating the new wire. This guarantees that after `onceE` has terminated the new wire will be told; it will tell all its termination listeners; and will also perform the necessary clean-up work. Note that because listeners are executed in FIFO order, when event `e` generates an occurrence, the input listener of `wire w`, and therefore in turn all of `w`'s listeners, will be fired *before* the termination listener is fired.

```
onceE :: Event a -> IO (Event a)
onceE NeverE = return NeverE
onceE e = do

 -- make a new termination action
 (tellterm,tnew) <- mkTermE

 -- as before create a new wire, to cache the single event -
 -- occurrence, only now we also pass in the new termE
 w <- primNewWire tnew
 addListener e (onceL (input w))

 onTerminate (termE e) tellterm
 addListener e (tellL () tellterm)

 return (event w)
```

Finally, we come to the definition of `switcherE`. Recall that this is also now an IO based definition. The semantics of switcher termination say that a '`switcher e ee`' will terminate when `ee` has terminated, and the current event has terminated. We therefore need to keep track of which is the current event, and whether the event `ee` has terminated.

A naïve implementation might assume that we could use one boolean variable to record whether the event `ee` has terminated, make `ee` set this when it terminates, and then add a termination listener to the current event that first checks this variable and only terminates when it is `True`. Unfortunately, here we come up against our fundamental termination assumption; that a termination event will only fire once. We cannot safely guarantee that `ee` will terminate before the last (current) event. For instance, if the last event were to be a `NeverE`, then it would clearly have already terminated. The termination event for `ee` would therefore fire later.

Instead we must use one variable (`curref`) to store an action that says whether the current event has terminated. When the switching event terminates, it checks whether the current event has terminated (using this variable), and if it has, it fires the new terminate action. When the current event terminates, it performs a similar action (this time checking the switching event using `isTerminated`).
On every occurrence of the switching event, we check the occurrence event. We set the current termination action to that of this new event, and register interest in its termination event. Finally, we must unregister interest in the previous event.

```
switcherE :: Event a -> Event (Event a) -> IO (Event a)
switcherE e NeverE = return e
```

```
switcherE e0 ee = do
  (tellt,tnew) <- mkTermE

  -- as before, except now using the termination event
  w <- primNewWire tnew
  addListener ee (switchL e0 (input w))

  currref <- newIORef (isTerminatedE e0)

  let -- when the switching event terminates, check if the current
      -- event has also terminated, and if so terminate
      actee = liftL1 $ \op v -> do
                b <- readIORef currref >>= id
                when b (op v)

      -- when the current event terminates, check if the switching
      -- event has terminated, and if so terminate
      acte = liftL1 $ \op v -> do
                b <- isTerminatedE ee
                when b (op v)

  -- initially we're interested in termination of e0
  rm1 <- onTerminateE e0 (acte tellt)
  removerRef <- newIORef rm1

  -- we're also always interested in termination of ee
  onTerminateE ee (actee tellt)

  -- whenever ee has an occurrence,switch interest to new event
  let f e = do
       writeIORef currref (isTerminatedE e)
       rmnew <- onTerminateE e (acte tellt)
       callRemover removerRef rmnew

  addListener ee (mkL f)
  return (event w)
```

7.2.6.3. Implementing listener-level switchers

Given the event termination combinators, we can now create a listener switcher. It behaves as follows. It starts by behaving as `l`; on every occurrence of `ee`, it hears of a new listener, and instead behaves like it. Recall that a listener has the power to remove itself from its source. There are important restrictions on when it is correct to do this. These are very similar to the event-level switcher rules above. The current listener should only be free to remove itself when the switching event has terminated. Also when the switching event terminates, it should check whether the current listener has tried to delete itself, and if so it should delete the whole listener.

We can implement this correctly using event termination. We use a variable to store the current listener's information. This contains either `(Just (rm,op))` if there is a current listener, where the pair represents the listener's own remove action (`rm`), and consumer action (`op`). The current listener is passed a remove action that checks whether the switching event has terminated. If so it removes itself; if not it sets the current listener variable to `Nothing`. When the switching event terminates, if the listener variable contains `Nothing` we perform the remove action. On every event occurrence, we change the current listener information. We store the new listener remove and consume action, and then perform the remove action for the old listener, to stop it doing any extra work. The consume action for this composite listener, consists of reading the current `opvar`, and applying the current operation to its argument, when there is a valid listener contained inside. Finally, we return remove actions to delete the terminate and event listener, along with the actual operation.

```
switcherL :: Listener a -> Event (Listener a) -> Listener a
switcherL l NeverE = l
```

```
switcherL l ee = mkPrimL $ \rmReal -> do
 opvar <- newIORef Nothing

 let dieL = do {b <- isTerminatedE ee;
                if b then rmReal else writeIORef opvar Nothing}

     dieEE = do {mb <- readIORef opvar;
                 when (not (isJust mb)) rmReal}

 val <- getPrimListener l dieL
 writeIORef opvar (Just val)

 let f l = do
      val' <- getPrimListener l dieL
      mb <- readIORef opvar
      maybe (return ()) fst mb
      writeIORef opvar (Just val')

 rmEE <- addListener ee (mkL f)
 rmT <- onTerminateE ee (mkL_ dieEE)
 let tell occ = do {mb <- readIORef opvar;
                    maybe (return ()) (($ occ) . snd) mb}
 return (rmEE >> rmT,tell)
```

This approach results in the correct semantics. For instance, consider the following two examples. In the first case we have a switching listener, that only allows each of its current listeners to fire once. In the second case, we have a switching listener that has `onceL` applied to it guaranteeing that the whole listener will only be fired once.

```
switcherL (onceL l1) (ee ==> onceL)
onceL (switcherL l1 ee)
```

In the first case each current listener will be passed a modified remove action that prevents it from removing itself, until the whole switcher has finished. In the second case each individual listener will be passed a modified remove action, but the complete `onceL` listener will be passed the full remove action, along with the delete action to stop the `switcherL` from doing any further work.

The use of event termination therefore provides a powerful mechanism for both implementing efficient event chaining, and to allow the correct and efficient implementation of other combinators such as `switcherL`.

7.2.7. Data Driven Behaviors

7.2.7.1. The Basic Behavior Definitions

We now come to the implementation of efficient data driven behaviors[25]. The implementation of behaviors is built on top of the simpler notion of a "sampler", which is a sampling function that returns a value and a boolean flag saying whether the sample value is at least temporarily constant.

```
type Sampler a = Time -> IO (a,Bool)
```

A behavior is something that can be sampled, and that has an invalidation event, that occurs whenever a constant segment ends.

```
data Behavior a = Behavior {sample :: Sampler a,
                            invalidateEv :: Event ()}
```

A constant segment is a period of time for which a behavior has a constant value. For instance, consider the definition `'0 `stepAccum` e -=> inc'`. Initially, it has the value `0`, then `1` after `e`'s first occurrence, then `2` and so on. Between occurrences of `e` this value does not change and so we have a constant segment.

[25] This representation is due to Conal Elliott.

How can we tell whether a behavior will be temporarily constant? It will be, if it is: (1) a constant behavior, (2) a reactive behavior (dependent on an event), and its current behavior is piecewise constant; (3) a function application of a behavior to another behavior, where both are piecewise constant. In contrast, a behavior based on the `time` primitive will not be piecewise constant.

Using this representation we can provide simple definitions of constant and time behaviors. A constant behavior always returns the same value. The `time` behavior returns the current time as its value, and will clearly not be constant[26]. Neither depend on an event in their definition.

```
constantB :: a -> Behavior a
constantB a = Behavior (\_ -> return (a,True)) neverE

time :: Behavior Time
time = Behavior (\t -> return (t,False)) neverE
```

We can define behavior function application "`$*`", which combines a function-valued behavior and argument behavior as follows. It combines the sample values, and the constant flags. As noted earlier the composition will only be constant, when both behaviors are constant. The invalidation event of the pair contains the merged invalidation event of each.

```
($*) :: Behavior (a -> b) -> Behavior a -> Behavior b
fb $* xb = Behavior sampler invalidate
 where
   sampler t  = do (f,fConst) <- sample fb t
                   (x,xConst) <- sample xb t
                   return (f x, fConst && xConst)
   invalidate = invalidateEv fb .|. invalidateEv xb
```

We can define the n-ary lifting functions in terms of this operator.

```
lift0 = constantB
lift1 f b1 = lift0 f $* b1
lift2 f b1 b2 = lift1 f b1 $* b2
```

These combinators are used to define many lifted functions. Some functions, especially non-strict ones, require special treatment. For example, a lifted conditional combinator might instead be defined as shown below. Note that it optimises both by only sampling the left arm of the conditional when the guard is True, and similarly for the right arm and False. The invalidate event again restricts occurrences, based on the current value of the guard.

```
condB :: Behavior Bool -> Behavior a -> Behavior a -> Behavior a
condB c b b' = Behavior sampler invalidate
 where
 sampler t = do (cVal,cConst) <- sample c t
                (vVal,vConst) <- sample (if cVal then b else b') t
                return (vVal, cConst && vConst)
   invalidate = invalidateEv c .|. invalidateEv b `whenE` c
                .|. invalidateEv b' `whenE` (notB c)
```

7.2.7.2. Behavior Caching

This representation has a problem. It redundantly samples behaviors that are used more than once. As in [46] we use a caching mechanism to avoid redundant computation. The important question is, how much caching is enough? If, as with events, we assume no notion of history, then we can get a simple efficient implementation, by caching only a single sample. We assume that sampling will take place at strictly increasing time intervals, and in particular, that it is only safe to sample a behavior at the current time. This unfortunately makes it impossible, for instance, to define a generalised time

[26] The existence of the `time` behavior here is, strictly speaking, unecessary for FranTk, as time behaviors are provided via `timeTick` and `timeTickNow`. However, its definition is given here as it needed by Fran.

transform. This restriction will be explored further when discussing reactive behaviors in Section 7.2.7.4. We cache behaviors only when really required, as in `"$*"` or `condB`.

A Cache therefore consists of an IORef with three values. A time representing the last time that the cache was updated, a boolean value determining whether the cache contains a value from a constant segment, and the actual sample value.

```
type BCache a = IORef (Time, (a,Bool))

newBCache :: IO (BCache a)
newBCache=newIORef (minTime,(error "undefined cache value",False))
```

The basic mechanism to sample a cache is the caching sampler function. This takes a cache and a sampler, and yields a caching sampler.

```
cachingSampler :: BCache a -> Sampler a -> Sampler a
cachingSampler cache sampler = \t -> do
  (tLast, p@(_, isConst)) <- readIORef cache
  if isConst || tLast==t then return p
   else do p <- sampler t
           writeIORef cache (t,p)
           return p
```

When we sample the cache we know we can use the currently cached value, either when it contains a constant value, or when the current sample time is equal to the last sample time (as in this case a time based behavior will return the same value).

We make a caching behavior from a sampler and invalidate event. At each invalidation, we check the cache. If it contains a constant value, we replace the sample time by the invalidation time and mark the cache as non-constant. There is a complication here. An event does not affect a behavior until immediately after the event occurrence. To handle this we clear the behavior cache just after the event, because other samples may occur between the invalidate event firing, and the actual change of the behavior. These samplings could set the cache back to constant, before the value of the behavior has changed, resulting in an incorrect result. We also assume that the time when the cache is sampled, is strictly later than the time when the cache is reset, otherwise the cache's value will still not be updated. We can achieve this by updating the cache with the time when the invalidating event was fired, not when the delayed action is run. Finally, when created we must also add a delayed action that resets the cache to non-constant. This is because the same event that is used to invalidate the cache may also create a caching behavior. Clearly in this case the cache's value should change when the behavior changes. Unfortunately, the cache will have missed this occurrence of the invalidation event. If the cache is sampled before its behavior is updated, then we will again have an incorrect value in the cache. Adding the delayed action therefore solves this problem.

```
cachingBehaviorIO :: Sampler a -> Event () -> IO (Behavior a)
cachingBehaviorIO sampler invalidate = do
   cache <- newBCache
   let reset cache _ = do
        te <- getTime
        addDelayedAction $ do
         (tLast, (x, isConst)) <- readIORef cache
         when isConst $ writeIORef cache (te, (x, False))
   addListener invalidate (mkL (reset cache))
   reset cache undefined
   return $ Behavior (cachingSampler cache sampler) invalidate
```

This implementation relies on support for delayed actions. We need to be able to add an action and guarantee that it will be performed in the future, after the current primitive event occurrence. More will be said on delayed actions in Section 7.2.7.6.

```
addDelayedAction :: IO () -> IO ()
```

We can define a pure function version of this caching action using `unsafePerformIO`. Note that it is safe to use this definition here, as the caching action will generate the same result irrespective of when it is evaluated. It will always generate a new sampling action that returns the current value of the cached behavior.

```
cachingBehavior :: Sampler a -> Event () -> Behavior a
cachingBehavior sampler invalidate =
  unsafePerformIO $ cachingBehaviorIO sampler invalidate
```

We can now make any function exploit caching simply by replacing any call to `Behavior`, with a call to `cachingBehavior`.

7.2.7.3. Snapshotting Behaviors

The existence of a sampler action in a behavior makes it easy to define snapshot for events and listeners. To snapshot a behavior on a listener, we simply sample the behavior at the occurrence time, and then return a pair of values containing the occurrence value and sample value. We can then define the event `snapshotE` combinator in terms of this listener combinator. Note that we cache the resulting event, to prevent unnecessary work. Otherwise the behavior would be sampled on every event occurrence, for every listener, rather than just once per occurrence. This caching is safe here as it only changes the efficiency of the combinator, not its meaning.

```
snapshotL :: Behavior b -> Listener (a,b) -> Listener a
snapshotL b = liftL1 $ \op (t,a) -> do
                (b,_) <- sample b t
                op (t,(a,b))

snapshotE :: Event a -> Behavior b -> Event (a,b)
e `snapshotE` b = cache (mapLE (snapshotL b) e)
```

7.2.7.4. Reactive Behaviors

Reactive behaviors are based on the `switcherB` primitive.

```
switcherB :: Behavior a -> Event (Behavior a) -> IO (Behavior a)
```

It can be implemented as follows. The implementation stores the sampler action for the current behavior in a variable. The current behavior changes on every event occurrence. In particular, the semantics of `switcherB` say that the behavior switches immediately after the event occurrence. There might, however, be samplings on the current time, still waiting to be done. We must therefore delay the sampler replacement with `addDelayedAction`. The behavior constructed by `switcherB` may have constant segments. We must define an appropriate invalidation event. There are two possible reasons to invalidate the behavior. The first is if the switching event occurs. The second is if the constant segment of the current behavior ends. We define the second using `switcherE`. This produces an event which behaves as the current invalidation event.

```
b0 `switcherB` e = do
  samplerRef <- newIORef (sample b0)
  let alter samplerRef b = do
        addDelayedAction $ do
         writeIORef samplerRef (sample b)
  addListener e (mkL (alter samplerRef))
  e' <- invalidateEv b0 `switcherE` (e ==> invalidateEv)
  let invE = (e -=> () .|. e')
  let sampler t = readIORef samplerRef >>= ($ t)
  return $ Behavior sampler invE
```

There is one obvious optimisation that we can apply to `switcherB`. If the switching event has already terminated or is `NeverE`, the result of `switcherB` will simply be the initial behavior.

```
switcherB b0 NeverE = return b0
switcherB b0 e = do dead <- isTerminatedE e
                    if dead then return b0 else ...
```

This simplification shows why the definition of switcherB must be IO based. The switcherB primitive depends on the event history to know which is its first occurrence. However, in the current implementation events do not store any history. It therefore matters when we evaluate switcherB. As with the history based event combinators we can define this IO based switcher in terms of the Elliott-Hudak FRP semantics using afterTime.

```
switcherB b e = do
  t <- getTime
  return (FRP.switcherB b (afterTime e t))
```

This change has a knock-on effect on several of the combinators discussed in the previous Chapter. The constructors to make behavioral collections all rely on `switcherB`. These must therefore all become IO (or GUI) based actions. Though an inconvenience, this is not a major "crisis", as most of the time we create behaviors using BVars. These are therefore already GUI based actions.

Though we can define stepper in terms of `switcherB`, we can in fact define a more efficient implementation. This implementation relies on the knowledge that the behavior will be piecewise constant, only changing when its switching event occurs.

```
stepper :: a -> Event a -> IO (Behavior a)
x0 `stepper` e = do
  samplerRef <- newIORef x0
  let alter samplerRef x = do
     addDelayedAction $ do
       writeIORef samplerRef x
  addListener e (mkL (alter samplerRef))
  let sampler t = do
        x <- readIORef samplerRef
        return (x,True)
  return $ Behavior sampler (e -=> ())
```

7.2.7.5. Delayed Actions

The semantics of `switcher/stepper/untilB` says that the behavior changes immediately *after* the occurrence time. In order to implement this semantics robustly, we don't update the behavior right away. Instead we add the update IO action to a collection of "pending actions". For each "primitive event" occurrence, we execute and clear the pending actions. For this to work, we must guarantee one important property. All of the delayed actions in the collection must be independent. We therefore need not assume any special ordering. The only time that we add delayed actions is to update a behavior variable, and to set a cache to non-constant. Neither of these actions involve sampling another behavior, or event and they are therefore safely independent.

7.2.7.6. Introducing Workpools

To efficiently exploit behaviors, we need to perform some actions as often as necessary, rather than a pre-determined number of times. For instance, when an aspect of a GUI element's display depends on a behavior, we only want to update the display when the behavior is updated, rather than after every time sample. To express this notion, we extend the listener idea to include adding a listener to a behavior.

To allow this to work we need an interface like the one below. We can add a listener to behavior, and run every behavior.

```
addListenerB :: Behavior a -> Listener a -> IO Remover
runIOBsOnce :: IO ()
```

For this to work, the listener's actions must be idempotent. This assumption is critically important, since it allows the implementation to be efficient in the common case that a behavior being output is temporarily or permanently constant. This seems a reasonable restriction because the timing or number of executions cannot be specified, and so non-idempotent actions will have non-deterministic results.

Since we have listeners on events and behaviors (as well as CVars and Channels in the concurrency interface discussed in the Section 4.14), we can introduce a type class to simplify the name space.

```
class HasListener c where
  addListener :: c a -> Listener a -> IO Remover

instance HasListener Behavior
instance HasListener Event
```

To implement addListener we maintain a pool of work items, each of which contains a sampler and a variable. The variable stores the remove action, to delete the item from the pool. In particular, this stores a Maybe value, which will contain the remove action only when the item is in the pool.

```
data WorkItem = WorkItem {
  removerRefW :: (IORef (Maybe Remover)), -- remove from workpool
  tiob :: (Sampler (IO ()))   -- to do
  }

type WorkPool = MutSet WorkItem
```

The Workpool is traversed as frequently as possible. For each item, the IO valued sampler is sampled, and the action is invoked. If the behavior is currently constant, the item is removed from the work pool, otherwise it is kept.

```
doWork :: Time -> WorkPool -> IO ()
doWork t pool = do foreachInMutSet pool (doOne t)
                   return ()

doOne :: Time -> WorkItem -> IO ()
doOne t (WorkItem remv sample) = do
 mb <- readIORef remv
 case mb of
   Nothing -> return ()
   Just rm -> do {(io,const) <- sample t;
                  io;
                  when const (do {rm;writeIORef remv Nothing})}
```

When adding a behavior listener, we first get the listener operation passing in the complete remove action to end all of the item's work. We create an IO valued sampler. It samples the behavior and returns the action to fire the listener with the current behavior value. We make a workpool item, and first restore the item to the workpool. We also add a listener to the invalidation event. This will restore the item to the workpool whenever a constant segment ends. When restoring an item we first check whether it is already in the pool. We only add it if it is not.

```
type WorkPool = MutSet WorkItem

addWorkItem :: WorkPool -> Behavior a -> Listener a -> IO Remover
addWorkItem pool b l = fixIO $ \allrm -> do
  op <- getListener' l allrm
  let sampler t = do {v,c) <- sample b t;return (op (t,v),c)}
  removerRef <- newIORef Nothing
  let item = WorkItem removerRef sampler
  let restore = restoreWorkItem (addToMutSet pool item) removerRef
  restore
  remove <- addListener (invalidateEv iob) $ mkL_ restore
  return $ do {mb <- readIORef removerRef;maybe (return ()) id mb;
               writeIORef removerRef Nothing;remove}

restoreWorkItem :: IO Remover -> IORef (Maybe Remover) -> IO ()
restoreWorkItem add removerRef = do
 mb <- readIORef removerRef
 when (not (isJust mb)) $ do
  {rm <- add;writeIORef removerRef (Just rm)}
```

The important remaining question is how frequently do we need to run the workpool? A Fran animation will have many behaviors based on time, and it is important to update the animation image regularly. It is therefore clearly important to have a timer that runs the workpool very regularly.

However, in a FranTk program it is far less common to include time based behaviors. This is because most GUI programs do not contain animations or other components that must be updated at a given rate; instead they simply react to user input. In addition, those programs that do rely on time will often only need to be refreshed at a significantly lower rate. We therefore need an alternative approach. The option adopted in FranTk is to run the workpool after every primitive event. A primitive event is, for instance, a mouse click or keyboard event. FranTk supports a global timer that will run only when there are non-constant behaviors in the workpool. The rest of the time this timer will be deactivated. We can set the refresh rate using the `setRefreshRate`. The GUI monad maintains in its environment the current refresh rate, and this is used when running the pool.

```
setRefreshRate :: Time -> GUI ()
```

As an alternative, recall that FranTk supports the `timeTick` function. This creates a time behavior that is refreshed every `t` seconds. This behavior will have its own, private timer. This provides a more efficient mechanism if we need to have several behaviors each with significantly different refresh rates, as they will all be updated at their own individual refresh rate, rather than simply at the fastest one.

```
timeTick :: Time -> GUI (Behavior Time)
```

7.2.8. Eliminating Work with Weak References

The implementation discussed so far is reasonably efficient in terms of time-performance, as it uses a data driven update mechanism. It does, however, still suffer from a number of time and space-performance problems. We noted that event-caching can result in space leaks as we end up with chains of events that will not be broken down. The event termination mechanism can solve one class of these space leaks, by breaking down a chain from the server end. However, there is another class that still needs to be dealt with. We need to be able to break down a chain from the client end. The problem with our representation is that a listener will keep talking to the event it is driving, even after it has no listeners and there is no longer a reference to the event.

The implementation of behaviors suffers from a similar problem. Consider the behavior `'b0 `switcherB` e'` in which `b0` and each new behavior generated by `e` is itself reactive. Whenever we lose interest in a reactive behavior, assuming there are no more references to it, the state variable it contains is no longer useful, nor is the work it takes to update it, which may come from a chain of events. However, the event chain side will contain a reference to the state variable, so it will not be reclaimed, and neither will the event chain.

All of this unnecessary work results in both a time-leak, because we will have many event chains redundantly firing, and a space-leak, because these event chains remain redundantly in existence. We need some mechanism to prevent unnecessary work by breaking down event chains when they have no more real subscribers.

We can solve these problems using *weak pointers* and *finalisers* [159]. These concepts are also present in other garbage collected languages such as Java. A *weak pointer* is a second class reference to an object, that does not keep the object alive. When all first-class references to an object are garbage collected, the object will therefore be garbage collected, irrespective of whether there are any weak references. A *Finaliser* is an action that will be run when an object dies. Haskell supports the following interface for using weak pointers[27].

---

[27] This interface is supported by GHC and Hugs. There is in fact a subtle problem with this interface. Standard weak pointers can have problems in GHC, because the *thunks* that weak pointers are using as keys can get optimised away. For instance, if `readIORef` is partially applied to an `IORef`, then the box that is holding the ref cell is discarded, leaving only the primitive ref cell. The weak pointer therefore dies as the `IORef` it was pointing to is gone. Instead GHC supports a function `mkWeakIORef`, which is a specialised version, of `mkWeakPtr` that works for `IORefs`. The key to the weak pointer is the primitive mutable ref cell inside the `IORef` box. This cannot be optimised away and the IORef will therefore stay alive. For simplicity, we will, however, just use `mkWeakPtr` in the following discussion.

```
mkWeakPtr     :: a -> Maybe (IO ()) -> IO (Weak a)
deRefWeak     :: Weak a -> IO (Maybe a)
addFinalizer :: a -> IO () -> IO ()
```

The function `mkWeakPtr` creates a value of type `Weak a`, which is a weak pointer to an item. It also takes a `Maybe` finaliser, of type `IO ()`. The `deRefWeak` function de-references a weak pointer, turning it into `Just a`, if the value is alive, and `Nothing` otherwise. The `addFinaliser` function simply adds a finaliser to an object. We can use this interface to implement truly efficient event-chains.

One potential implementation of `mkWire` might work as follows. Instead of making a listener talk directly to an event, we instead make it talk to its listener set via a weak reference. This guarantees that the listener set only stays alive as long are there is a reference to the event. Unfortunately, this simple model has a problem. Consider the following code:

```
main _ = do
  w <- newWire
  addListener e (mkL print)
  performGC
  fire <- getListener l
  fire (1,"1")
```

This will print nothing. This is because once we have added the listener, we drop the reference to `e`, and then garbage collect. As `l` does not keep the listener set within `e` alive, the listener set is garbage collected and so nothing happens. Instead we need to ensure that the listener keeps its listener set alive as long as there are elements in the listener set. Is this important. Yes! Significantly, it arises when we add a listener to a behavior. The work item is removed from the pool when it is constant. Then the only reference to it is in the listener on the item's invalidate event.

We therefore need an alternative event finalisation semantics. A wire with listener (l) and event (e) will sustain its listener set according to the following rules.

1. An event remains alive as long as its internal listener set is alive.
2. When `l` remains alive, (i.e. there is something that can talk to the event), and there are any listeners to `e`, or there is a reference to `e` (that could therefore add a listener in the future), the listener set remains alive.
3. If `l` is alive, but there are no listeners and there is no reference to `e`, the listener set dies.
4. If `l` dies (i.e. there ceases to be anything that can talk to the event) then the listener set dies, and the event terminates.

Recall that listeners can hear about their own remove actions. We can use these remove actions to unregister a wire's listener from all its sources when its event dies. We can therefore break down event chains from the client end using this mechanism.

The wire's listener talks to the listener set via a variable. Its reference can be either `Passive`, `Active` or `Dead`.

```
type ListenerSet a = MutSet (Occ a -> IO ())
data Status a = Passive | Dead | Active (ListenerSet a)
```

The listener therefore only sustains the listener set when it is actually active. When the listener end (i.e. the listener variable) dies, we run a finaliser that kills the listener set.

The event end has a direct reference to the listener set. It therefore sustains it as long as the event is alive. The event end also maintains a weak reference to the listener end. The finaliser for the listener set uses this reference to set the listener end to `Dead`. When the listener set is emptied, this reference is used to set the listener end to `Passive`. If the listener set is empty, and a listener is added, this reference will be used to set the listener end to `Dead`. This story is illustrated in Figure 41.

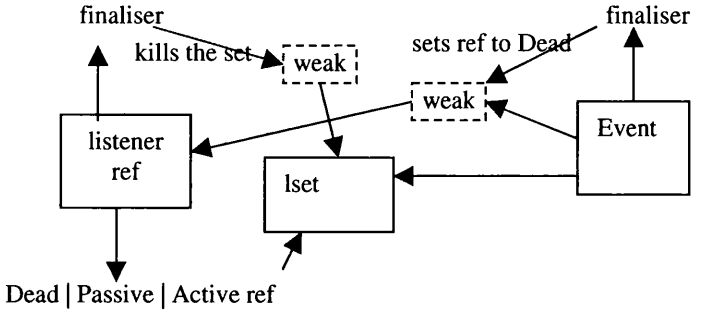

Figure 41 - Wire References with Weak Pointers

We need to refine this model a little. To handle event termination we instead make the central object a variable containing both the listener set *and the event terminate action.* This guarantees that the event terminate action is also only sustained as long as the event is alive. If all future references to it are lost, then the terminate event will not be sustained. This is necessary as the terminate event may end up with a direct reference to the actual wire's event. If the listener had a direct reference to the terminate event, the event would still be unnecessarily sustained. In this model the listener sustains this central variable, when it is active.

```
type MainRef a = IORef (ListenerSet a, IO ())
data Status a = Passive | Dead | Active (MainRef a)
```

We can therefore implement a new primitive wire constructor. This takes a finaliser action that will be performed when the event dies, and a termination event.

```
primNewWire :: IO () -> TermE -> IO (Wire a)
primNewWire final termE = do
```

As before we first create a mutable listener set for the wire.

```
  lset <- newMutSet
```

We also create a terminate event that will be fired when the event terminates.

```
  (tellterm,newtermE) <- mkTermE
  tellt <- getListener tellterm
  eref <- newIORef (lset,do {t <- getTime;tellt (t,())})
```

We create a kill variable that contains a `Maybe` value for all the remove actions for sources that the wire's listener has been added to. The actions in this kill variable will be performed when the event dies. Once the event has been killed this `Maybe ref` will be set to `Nothing`, thereby preventing any further kill actions from being added.

```
  killref <- newIORef $ Just (return ())
  let addKill k = updIORef killref (fmap (\old -> old >> k))
  let fireKills = do {mb <- readIORef killref;
                      writeIORef killref Nothing;
                      maybe (return ()) id mb}
```

Initially, the event has no listeners and so its listener reference is passive.

```
  tellref <- newIORef Passive
```

When the listener end or event end dies it has an action that will be run.

```
  eKillRef <- newIORef (return ())
  lKillRef <- newIORef (return ())
```

We make a weak reference to the listener end; its finaliser kills the event end when the `tellref` dies. Similarly, we make a weak reference to the event end (i.e. the listener set); its finaliser kills the listener end when the listener set dies.

```
lwp <- mkWeakPtr tellref (callRemover eKillRef (return ()))
ewp <- mkWeakPtr eref (callRemover lKillRef (return ()))
```

We also add a listener to the dependent termination event, that was passed in as an argument at the start. It will also kill the event end. Note that this again uses the weak reference so that the `termE` does not keep the central object alive. This is not now strictly necessary, because the wire's listener will die when the previous event terminates anyway. However, it can result in faster destruction of event chains.

```
rmT <- onTerminate termE
                    (mkL_ (do{callRemover eKillRef (return ())}))
```

When the event end dies, it fires the `lKillRef`. This sets the `tellref` to `Dead`; fires the kill actions, to remove the wire's listener from all its source events; runs the final action; and invalidates the `eKillRef`, as it no longer needs to be fired.

```
let alter :: (a -> IO ()) -> Weak a -> IO ()
    alter act wp = do
      {mb <- deRefWeak wp;maybe (return ()) act mb}

writeIORef lKillRef $
 do {alter (flip writeIORef Dead) lwp;fireKills;final;
     writeIORef eKillRef (return ())}
```

When the listener end dies it kills the listener set and also tells the termination event that the event has terminated. It runs the final action, removes the termination listener, and invalidates the `lKillRef`, as it no longer needs to be fired.

```
writeIORef eKillRef $
 do {alter (\ref -> do {(mset,tellt) <- readIORef ref;
                        killMutSet mset;tellt}) ewp;
     final;rmT;writeIORef lKillRef (return ())}
```

When we add an action to the event, we add it to the listener set. We also check if the listener set was previously empty, and if so we set the listener end to `Active`. If the listener set has already been killed (i.e. `sizeMutSet` returns `Nothing`) we do nothing.

```
let add :: Listener a -> IO Remover
    add l = do
       (lset,_) <- readIORef eref
       szMb <- sizeMutSet lset
       case szMb of
        Nothing -> return (return ())
        Just sz -> fmap snd $ fixIO $ \ ~(rm,_) -> do
                    (rmextra,fire) <- getPrimListener l rm
                    when (sz == 0) $
                     alter (\r -> writeIORef r (Active eref)) lwp
                    rm <- addToMutSet lset fire
                    rm <- mkRemover (remove rm)
                    return (rm,rm>>rmextra)
```

The `mkRemover` function makes a one-shot (idempotent) action from a simple action. We do this with remove. The remove function takes a remove action to delete an item from the mutable set and wraps it. When a listener is removed, we carry out the remove action. We also check to see if the listener set is now empty, and if so we set the listener end to `Passive`. Note that we use a weak reference to the listener set here so that this new remove action does not, itself, sustain the set.

```
remove :: Remover -> Remover
remove rm = do
  rm
  mb <- deRefWeak ewp
  case mb of Nothing -> return ()
             Just eref -> do
                (lset,_) <- readIORef eref
                mbSz <- sizeMutSet lset
                when (maybe False (==0) mbSz)
                     (alter (\r -> writeIORef r Passive) lwp)
```

To tell the event about an occurrence we must first check the listener end variable. If the reference is dead, or passive we do nothing. If it is active, we apply the occurrence. When generating the remove action we add the remover to the to the kill variable and then simply return the tell action.

```
let tell occ@(t,a) = do
     val <- readIORef tellref
     case val of Active eref -> do (set,_) <- readIORef eref
                                   foreachInMutSet set ($ occ)
                 _ -> return ()

let tellL = mkL' $ \rm -> do {addKill rm;return tell}

return (Wire tellL (Event add newtermE))
```

We now have a complete definition of `primNewWire`, that will break down event chains when they are no longer needed. Note significantly that we no longer need to pass in the termination event to the new event. This is because when its listener dies we know anyway that an event has terminated. It is still useful to return a termination event, as this is still necessary to define functions such as `switcherL`. However, the weak pointer approach subsumes the event termination mechanism, removing the necessity to perform extra work when defining the cached versions of `switcherE` and `onceE`. It can, however, be more efficient to support both approaches, as event termination will result in faster destruction than the weak reference approach.

We can refine this definition still further, by passing in a subscription and unsubscription function, when we create a new wire. Whenever the listener set is emptied we run the unsubscribe action. When a listener is added to an empty `lset`, we run the subscribe action.

This implementation will deal with event chains. We still, however, need to deal with unnecessary work in `switcherB`. We can do this by introducing an `addWeakListener` function. This adds a listener that remains only as long as its client (`IORef`) is alive. Note that the listener does not keep the client alive. When the client dies we delete the listener from the event. As we now have a general class of functions with listeners, the `addWeakListener` function simply works with members of this class.

```
addWeakListener :: HasListener c => c a
                -> (IORef b -> a -> IO ()) -> IORef b
                -> IO Remover
addWeakListener e lisf client =
  fixIO $ \ remLis -> do
    wp <- mkWeakPtr client remLis
    remLis <- addListener e $ mkL $ \x -> do
                mbClient <- deRefWeak wp
                case mbClient of
                  Nothing  -> remLis
                  Just client -> (lisf client) x
    return remLis
```

We can use this definition in the definitions of `switcherB`, `stepper` and `cacheBehavior`. All of these functions have a state variable, that can be used as the client. We can therefore easily replace the definition of `addListener`, with `addWeakListener`.

7.2.9. Summary

The data driven implementation discussed in this section is both space and time efficient. Events and behaviors are only updated when necessary, as changes are pushed from the source, rather than pulled as in the functional streams approach. The use of weak references prevents unnecessary work by forcing server listeners to stop talking to client events when they are no longer needed.

However, as we have seen, this implementation also has a fundamental referential transparency problem. As events maintain no history, we must redefine the types of any history based event or behavior combinator, to work in the IO (or GUI) monad.

7.3. An Efficient Hybrid Solution?

This section will present a hybrid implementation that combines the Elliott-Hudak functional streams implementation with the data-driven implementation. This hybrid is faithful to the Elliott-Hudak formal semantics. However, it has one unfortunate robustness problem, which will be discussed as we go along.

7.3.1. Implementing Events

7.3.1.1. Defining events

To provide a correct implement of events we merge the functional streams representation with the data-driven representation.

```
data Event a = NeverE | Event [PossOcc a] GetLastOcc InvE
```

An event is now considered to consist of three parts.

1. A stream of possible (`Maybe`) occurrences. This stream is a lazy list implemented in a similar manner to the streams representation.

   ```
   type PossOcc a = (Time, Maybe a)
   ```

2. One problem with the streams approach is the need for events to have non-occurrences for each and every sample time. We overcome this by giving an event an IO action that returns information about the latest occurrence (`LastOcc`). This contains two values. The first is the time of the latest occurrence. An event may end up having several occurrences in the same time interval. The second value therefore specifies how many occurrences there have been at that time. When reading from an event stream, we therefore need to get the last occurrence information. It is only safe to read from the stream up until the last occurrence. To allow this to work, we must guarantee that an event has at the very least, once possible occurrence at the given occurrence time. This means that we may need to pad the event with at most one non-occurrence at time `lastocc`.

   ```
   data LastOcc = LastOcc Time Int
   type GetLastOcc = IO LastOcc
   ```

3. Finally, an event contains a invalidation event, that is simply a data-driven events as discussed in the previous section. Note that this no longer contains any information about the current value of an event.

   ```
   type InvE = DataDriven.Event ()
   ```

By merging both concepts we end up with a representation that contains information about how an event should be used (the `InvE`) and about its semantics (the stream of possible occurrences).

7.3.1.2. Implementing some basic combinators

With this implementation we can implement many of the event combinators using the streams approach. For instance, we can implement `mapE` as shown below. This maps its function argument across every real occurrence in the list.

```
mapE :: (a -> b) -> Event a -> Event b
mapE _ NeverE = NeverE
mapE f (Event possOccs get inv)=Event (map apply possOccs) get inv
 where apply (te,Just a) = (te,Just (f a))
       apply (te,Nothing) = (te,Nothing)
```

We reintroduce the generalised mapping combinator, `handleE`. The function argument takes the occurrence time, the value, and the rest of the event after that occurrence.

```
handleE :: Event a -> (Time -> a -> Event a -> b) -> Event b
handleE NeverE _ = NeverE
handleE (Event possOccs get inv) f = Event (loop possOccs) get inv
 where loop [] = []
       loop ((te, mb) : possOccs') =
        (te, fmap (\x -> f te x (Event possOccs' get inv)) mb)
          : loop possOccs'
```

Similarly, because we have access to the occurrence stream, we can use a purely stream based implementation to implement `scanlE`.

We can implement `mapMaybeE` in terms of the streams implementation. However, again it is more efficient to cache it.

```
mapMaybeE :: Event a -> (a -> Maybe b) -> Event b
mapMaybeE NeverE _ = NeverE
mapMaybeE (Event occs get inv) p =
  cache (Event (filterStream p occs) get inv)
```

7.3.1.3. A problem with mergeE

We come across an immediate difficulty when trying to implement `mergeE`. The purely stream based implementation is shown below. Unfortunately, this definition is one of a number that relies on the existence of non-occurrences at every sample time. When checking for a possible occurrence on one stream, the definition assumes the existence of an occurrence in the other stream. This restriction is obviously not satisfied by our new representation.

```
merge :: [PossOcc a] -> [PossOcc a] -> [PossOcc a]
merge [] os' = os'
merge os []  = os
merge os@(o@(te, _) : osRest) os'@(o'@(te', _) : osRest')
      | te  <=  te' =  o : merge osRest  os'
      | otherwise   =  o' : merge os      osRest'
```

To solve this problem, we must use a mixture of the two combinator approaches. We first get the event history from both events. We create a new wire, with an initial history formed by merging both event histories, and a last occurrence time. This is formed by merging the two last occurrence values. Merging takes the latest of the two occurrences. When both values occur at identical times we add their occurrence numbers together. We then add a listener to both of the argument events, that talks to the merged event.

```
mergeE :: Event a -> Event a -> Event a
mergeE e1 e2 = unsafePerformIO $ do
 (lst1,occs1,e1') <- afterCurrent e1
 (lst2,occs2,e2') <- afterCurrent e2
 w <- primNewWire (merge occs1 occs2) (mergeLast lst1 lst2)
                  (mergeTerm e1 e2)
 addListener e1 (input w)
 addListener e2 (input w)
 return (event w)

mergeLast lst1@(LastOcc t1 n1) lst2@(LastOcc t2 n2) =
  if t1 == t2 then (LastOcc t1 (n1+n2))
  else if t1 > t2 then lst1
  else lst2
```

This definition has one important semantic problem. The stream merge function guarantees that if two events have an occurrence at the same time, then the occurrence in the left stream will happen before the occurrence in the right stream. Unfortunately, the situation is not so simple for the data-driven listener implementation. In a simple approach, without event caching, we *can guarantee* this ordering, because the Fran time changes on each primitive event. The only way to generate two occurrences with the same time, is to generate two events that are functions of the same primitive event. For instance, we could define `e .|. (filterE even (mapE f e))`. Any listener added to this merged event will first be added to `e` and then to the filtered version of `e`. The listener will therefore be fired for `e` before the filtered `e`, as listeners are executed in FIFO order. Unfortunately, in the presence of caching this becomes infinitely more complex. For instance, the above example will generate the event chain shown below. If we assume that `filterE` has already been evaluated we end up with the following situation: the `filterE` listener is added to `e`; then the `mergeE's` listener is added to `e` and then to `filterE`. When `e` is fired, it will first tell `filterE`, then the merge; then `e` will tell the left arm of the merge. This clearly results in an inversion of the desired effect, because the right arm of the merge hears an occurrence before the left arm.

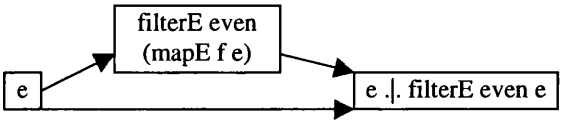

Figure 42 - Event Propagation in the Presence of Caching

In general, with the current data-driven event representation, it seems impossible to correctly order event occurrences. Unfortunately, it also seems impossible to recall the actual ordering of the semantic data-driven actions using the stream of occurrences. We therefore have our first fundamental problem with this implementation. A merge event will be unsafe when (1) on merging, both events have a history; (2) in each history, both events have an occurrence at an identical time; (3) the merged event is then passed to a function that uses the history, and cares about the ordering. Fortunately, this set of circumstances has arisen only rarely in the FranTk examples that have been tried so far. However, the restriction is unfortunate and unpleasant as it forces the programmer to carefully consider any use of the merge combinator.

As an aside, it is interesting to note that while Elliott and Hudak defined a semantics for such simultaneous occurrences, Wan and Hudak do not. They state that the results of a simultaneous occurrence with merge are non-deterministic. As a result, the non-determinism in the implementation presented here does not violate their semantics.

7.3.1.4. Creating a wire

We can define the wire constructor using the data driven occurrence mechanism. We create an imperative, data driven wire. We also create a channel to maintain the lazy stream, and a variable to store the time of the last occurrence. We modify the listener returned by the imperative wire. It now tells the channel about its occurrence value. It increments the latest occurrence variable, and then tells the imperative wire to fire all its listeners. As this wire may be based on an event with a history already, we pass in an initial latest occurrence time and list of initial values.

```
primNewWire :: [PossOcc a] -> LastOcc -> TermE -> IO (Wire a)
primNewWire occs last terme = do
 chan <- newChan
 lastref <- newIORef last
 let tellChan (t,a)= do writeChan chan (t,Just a)
                        (LastOcc t' n') <- readIORef lastref
                        writeIORef lastref $
                           if t == t' then LastOcc t (n'+1)
                           else LastOcc t 1
 (DataDriven.Wire l inve) <-
    DataDriven.primNewWire (return ()) terme
 let l' = liftL1 (\op occ@(t,_) -> do {tellChan occ;op (t,())}) l
 cs' <- getChanContents chan
 let cs = occs ++ cs'
 return (Wire l' (Event cs (readIORef lastref) inve))
```

We define an `afterCurrent` function that returns three values, the data about the last occurrence time, the list of occurrences that have happened before now, and a new event after all these occurrences have been stripped. It applies the `splitOccs` function which attempts to split the occurrence list at the last occurrence time. This function returns three values: the occurrence time that the list was really split at (if the list terminated before the occurrence time it will return the terminated occurrence), and the lists before and after the split. We add one non-occurrence at that last time, as we have to maintain the invariant that there is AT LEAST one possible occurrence in the stream at all times. Note that `afterCurrent` is lazy in both the result of `getLast` and in the event's stream. This will be important when we come to dealing with snapshot.

```
afterCurrent :: Event a -> IO (LastOcc,[PossOcc a],Event a)
afterCurrent NeverE = return ([(0,Nothing)],LastOcc 0 1,NeverE)
afterCurrent e@(Event occs getLast inv)  = do
  l <- getLast
  let (l',pre',os') = splitOccs nm l os
  let get' = do
      last'@(LastOcc tl' n') <- get
      return $ if tl' == occTime l' then
                  (LastOcc tl' (n' - occNum l' + 1))
               else last'
  return (l',pre',Event ((occTime l',Nothing):os') get' inv)
```

To cache an event, we use a similar mechanism to that used for data driven events, except that we also access the previous occurrences.

```
cache :: Event a -> Event a
cache NeverE = NeverE
cache e@(Event occs get inv) = unsafePerformIO $ do
 (lst,occs,e') <- afterCurrent e
 w <- primNewWire occs lst (termE e')
 addListener e' (input w)
 return (event w)
```

To add a listener to an event, we check if it has terminated: if so we do nothing; if not we create a variable to store the current occurrence list, and previous occurrence time. Note again that adding a listener is not strict in the occurrence list. We modify the listener, using fire, and add it to the data-driven invalidation event. Every time we fire the listener, we check the current and last occurrence time. If they are the same, we do nothing. Otherwise, get the occurrence stream, and break it at the last occurrence. We then update the occurrence stream variable. Only when the occurrence is a real one (i.e. it has the value `Just a`), do we perform the listener's action. If the occurrence list has for some reason already terminated, we clean up and remove the listener.

```
addListener :: Event a -> Listener a -> IO Remover
addListener NeverE l = return (return ())
addListener e NeverL = return (return ())
addListener (Event occs getLast inv) (l :: Listener a) = do
 isdead <- isTerminatedE inv
 if isdead then return (return ())
  else do
  ref <- newIORef occs
  prevref <- getLast >>= newIORef
  let fire :: Remover -> Listener' a -> IO ()
      fire rm op = do
       last@(LastOcc t n) <- getLast
       last' <- readIORef prevref
       unless (last == last') $ do
        all <- readIORef ref
        case dropOccs (LastOcc t (n-1)) all of
         (t,Nothing):rest -> writeIORef ref rest
         (t,Just val):rest -> do{writeIORef ref rest;op (t,val)}
         [] -> do {writeIORef ref [];rm}
  DataDriven.addListener inv
              (liftLIO (\rm op -> return (\_ -> fire rm op)) l)
```

7.3.1.5. More complex combinators

We might consider trying to implement `onceE` by simply applying the `once` combinator. To allow event termination to work we also cache the event.

```
onceE :: Event a -> Event a
onceE NeverE = NeverE
onceE (Event occs get inv) = cache (Event (once occs) get inv)
   where once [] = []
         once (o:os) = o:if isJust (snd o) then [] else once os
```

We can implement `switcherE` using a combination of the data-driven switcher implementation and `afterSwitcher`. The `afterSwitcher` function should apply a stream based switcher function to the events' previous occurrences. It returns the last occurrence time, the list of possible occurrence, the latest current event and the remaining switch event.

```
afterSwitcher :: Event a -> Event (Event a)
              -> IO (LastOcc,[PossOcc a],Event a,Event (Event a))
```

7.3.1.6. Saving space

As this hybrid implementation has access to a complete event stream, it is often necessary to be able to generate a new event, with occurrences only after a given time. We can do this using `afterTimeE`. We drop all occurrences before the given time. We add one non-occurrence at time `t`. Until the last occurrence time becomes greater than t, we simply return that occurrence as the latest. This maintains the stream invariant that there is one possible occurrence at, or after the current `LastOcc` value.

```
afterTimeE :: Event a -> Time -> Event a
afterTimeE (Event occs get inv) t =
  Event ((t,Nothing):dropWhile ((<= t) . fst) occs) get
        (DataDriven.afterTimeE inv t)
   where
    get' = do last@(LastOcc t' n') <- getInvE inv
              return $! if t' > t then last else LastOcc t 1
```

This function is very important. It has two uses. Firstly, we may need it for semantic reasons. For instance, we may wish to create a behavior that showed the number of mouse clicks, after a given event. In the data-driven approach, such a behavior could only be created in the IO monad, and so would have an implicit `afterTime`. In contrast, with the hybrid implementation, as an event now has a history, we must do this explicitly. The second reason is for efficiency. Because we're hanging on to event histories, they will build up. This could again result in a space leak, if we redundantly retain a copy of the complete history. Fran uses a generalised class, `Ageable`, and a function `afterE` to overcome this. The event `afterE` simply pairs up event occurrences with aged values. It is always safer to remove event histories wherever possible using this function.

```
class Ageable a where
 afterTime :: a -> Time -> a

afterE :: Ageable bv => Event a -> bv -> Event (a,bv)
```

By judicious use of caching when implementing the event combinators we have guaranteed that the data-driven event will only fire when there is a real occurrence. Only `filterE`, `onceE` and `afterTimeE` will cause non-occurrences to appear. We have cached both of the first and the definition of `afterTimeE` applies both to its data-driven event, and its stream list. This guarantee is useful as it makes the implementation more efficient and helps when defining behaviors next.

7.3.2. Implementing Behaviors

We now come to the implementation of a behavior. One possible representation of a behavior is again to merge the stream based and data-driven implementation. However, because events are no longer guaranteed to have non-occurrences at every sample time, we cannot implement a simple declarative version of `switcherB`.

We instead use the following representation. A behavior contains a `BStruct` which represents the structure of the actual behavior, and a data driven behavior, which contains an imperative sampler and an invalidation event.

```
data Behavior a = BehaviorB (BStruct a) (DataDriven.Behavior a)

at :: Behavior a -> Time -> IO (a,Bool)
at (Behavior _ b) t = sample s t
```

The `BStruct` type represents the structure of the actual behavior. A behavior is either constant; time based; a `switcher`; a `stepper` or a function application of one behavior to another. The function application uses existential types[115]. The type `b` is existentially qualified in the constructor. The only thing we can do with a value of type `b`, is to apply the `(b -> a)` function to it. We can, however, deconstruct the behaviors themselves.

```
data BStruct a = ConstantB a
               | TimeB (Time -> a)
               | SwitcherB (Behavior a) (Event (Behavior a))
               | StepperB a (Event a)
               | forall b . AppB (Behavior (b -> a)) (Behavior b)
```

The definitions of `constantB`, `time`, and behavior function application (`$*`), can be defined in terms of the `BStruct` and data-driven implementations. We can perform a simple optimisation when applying (`$*`) to two constant behaviors. This will simply generate another constant behavior.

```
BehaviorB (ConstantB f) _ $* BehaviorB (ConstantB x) _ =
 constantB (f x)
```

When implementing `stepper` and `switcher` we use the event history, to decide which is the current value. The `afterCurrentMb` function simply returns a `Maybe` value, representing the last real occurrence if there was one, instead of returning the entire list of previous occurrences. If the event has terminated we can then simply return a constant sampler. Otherwise we perform as before. Note that because we have guaranteed that the imperative invalidation event will only fire when there is a real occurrence in the stream, we can safely use just it.

```
afterCurrentMb :: Event a -> IO (Maybe a,Event a)

stepperImperative :: a -> Event a -> DataDriven.Behavior a
stepperImperative x0 e@(Event _ _ inve terme) =
   unsafePerformIO $ do
     (mb,ev) <- afterCurrentMb ev
     let x = maybe x0 id mb
     samplerRef <- newIORef x0
     let alter samplerRef x = do
           addDelayedAction $ do
             writeIORef samplerRef x
     addWeakListener e alter samplerRef
     let sampler t = do
           x <- readIORef samplerRef
           return (x,True)
     return (DataDriven.Behavior sampler inv)
```

To implement snapshot we again unite the data-driven and stream based implementations. We create a new wire that the original event talks to. The new wire's listener snapshots the behavior. We perform a stream based snapshot of the event's previous occurrences with the behavior.

```
snapshot ev bh = unsafePerformIO $ do
    (t,occs,ev') <- afterCurrent ev
    let bs = unsafePerformIO $ sampleAts bh (map fst occs)
    w <- primNewWire t (snapStream occs bs) (termE ev)
    addListener ev' $ snapshotL bh (input w)
    return (event w)
```

This definition uses the `sampleAts` function. This samples the behavior at a list of times, by applying a stream based sampler to the `BStruct`. The sampler is an IO action because, when it comes across a switcher or stepper, it only looks for occurrences before the last occurrence time. This is therefore safe as it does not search the event streams for future occurrences. This function is therefore only safe if the maximum time in the time stream is before the time when `sampleAts` is applied.

```
sampleAts :: Behavior a -> [Time] -> IO [a]
```

The snapshot function is supposed to be lazy in its behavior argument. We can overcome this by applying the `sampleAts` function lazily, using `unsafePerformIO`. This is safe because the last occurrence time we sample against was before the `afterCurrent` action is evaluated. At any time after this the `sampleAts` function will therefore return the same result. The implementation of `snapStream` requires to be *completely lazy in bs*. If it is, then `bs`, and therefore the behavior, will only be evaluated when we actually evaluate one of the new occurrence values in the resulting `snapStream`.

This observation is very important. The snapshot function needs to be lazy to allow the safe definition of self-reactive, and mutually-reactive behaviors. For instance, we might have the following definition. Here we have two integer valued events and two behaviors. Each behavior depends on one of the events, and shows the last value of the other behavior. Clearly if snapshot was strict in its behavior argument, when evaluating one behavior, we would end up evaluating the snapshot event, which would end up evaluating the other behavior. This would therefore fail.

```
e1,e2 :: Event Int
b1,b2 :: Behavior (Int,Int)
b1 = (0,0) `stepper` (e1 `snapshot` fstB b2)
b2 = (0,0) `stepper` (e2 `snapshot` fstB b1)
```

Any combinator that is strict in the previous occurrence list will still result in unacceptable strictness. The combinators shown so far have all been lazy. (Recall that caching and `addListener` were both lazy). One problem exists with `switcherE`. We must add a listener to the current event, which will obviously depend on the switching event's occurrence list. We can fix this by delaying this `addListener` action, and making `afterSwitcher` lazy by using `unsafePerformIO`. Though this works, the solution is frankly a hack.

7.3.3. Summary

This section has presented a hybrid FRP implementation that attempts to be faithful to the Elliott-Hudak semantics. In this it largely succeeds. Unfortunately, it is not perfect. Under certain circumstances `mergeE` will not be referentially transparent. This implementation therefore represents a step forward in developing FRP implementations, rather than the end of the story. In particular, it is presented to demonstrate the difficulties in implementing a correct, robust and efficient implementation of FRP, and is intended to act as inspiration for future work in this area.

7.4. A Third Data-Driven Implementation?

So far we have seen two possible data driven implementations. The first satisfied neither of the FRP semantic models. The second attempted to satisfy the Elliot-Hudak semantics. A third alternative is to try to refine the data-driven implementation to satisfy the Wan-Hudak semantics. This section outlines a plan of how this could be done. Future work is required to thoroughly test and validate these ideas.

Recall that the Wan-Hudak semantics defines events and behaviors as functions of both a start time and a sample time. Recall also the effect this has on combinators such as `untilB`. As before the behavior `'b `untilB` e'` exhibits `b`'s behavior until `e` occurs. It then switches to the behavior associated with `e`. However, *the new behavior will start afresh at the time of e's occurrence*. This is very significant. It means that such combinators do not hold onto an event's history. This results in a strong similarity to the data-driven implementation discussed in section 7.2.

7.4.1. The Basic Definitions

An event is a function from `StartTime -> (Time -> Occurrences)`. We therefore have two parts. An `Event` is a function that takes a start time and produces a *started* event of type `E a`. Performing the IO action starts the event.

```
data Event a = Event (Time -> IO (E a))
```

A *started* event of type `E a` allows listeners to hear about occurrences. It is therefore a subscription function for listeners.

```
type E a = Listener a -> IO Remover
```

We therefore differentiate internally between a started event, and an unstarted event. The internal event will not see any occurrences before the start time.

We can use a similar mechanism for behaviors. A `Behavior` is function from a start time, that produces a *started* behavior of type `B a`.

```
data Behavior a = Behavior (Time -> IO (B a))
```

A *started* behavior of type `B a` is identical to our first data driven behavior definition (section 7.2.7). It contains a sampler action and an invalidation event.

```
data B a = B (Sampler a) (E ())
```

7.4.2. Basic Event Combinators

We can implement the event algebra in terms of a few basic event combinators. The most primitive event combinator is `mapIOFE`. It takes an IO valued function to be applied to a *started* event and start time, and an event and generates a new event.

```
mapIOFE :: (E a -> Time -> IO (E b)) -> Event a -> Event b
mapIOFE f (Event mk) = Event $ \t ->
    do e <- mk t
       f e t
```

We can define the `mapLE` function in terms of this. Recall that many of the event algebra combinators are defined in terms of this (section 7.2.3).

```
mapLE :: (Listener b -> Listener a) -> Event a -> Event b
mapLE f = mapIOFE (\add t -> return $ \l -> addL t add (f l))
```

The `mapLE` maps a listener combinator across an event. When we add a listener to the new event, we first apply the listener combinator to the argument listener, before adding the resulting listener to the internal event. This function makes use of `addL` to add the listener. This takes a start time, started event and a listener and makes the listener hear every event occurrence *after the start time.* This is important because the formal semantics says that an event produces a finite list of time-ascending occurrences in the interval `(T,t]`, where `T` is the start time. Occurrences at the start time are therefore ignored.

```
addL :: Time -> E a -> Listener a -> IO Remover
addL t e l = e (afterTimeL l t)
```

7.4.3. Stateful Event Combinators

There are a number of stateful event combinators that depend on an event's start time. For instance, `scanlE` is one of them. These were the combinators that caused problems with the first data driven implementation (see Section 7.2.5).

To define `scanlE` we need to create an `IORef` to store the current value of the event. We do this when the event is started. Then on every occurrence we apply `f` to the occurrence value and current value, and update the `IORef`. All listeners to the new event just read the value of this `IORef`. Note that because listeners are handled in FIFO order this is safe. The IORef will be updated before any dependent listener is fired.

```
scanlE :: (a -> b -> a) -> a -> Event b -> Event a
scanlE f a = mapIOFE $ \e t0 -> do
  ref <- newIORef a
  addL t0 e
         (mkL $ \b -> do  v <- readIORef ref
                          let v' = f v b
                          writeIORef ref $! v'
                          return ())
  return $ \l -> addL t0 e (comapIOL (const $ readIORef ref) l)
```

We can use a similar mechanism for other stateful event combinators such as `switcherE`.

7.4.4. Memoisation

There is a performance problem with this implementation of Events and Behaviors. If we define a event or behavior once and use it in two different places, the IO action to start it will be run twice. However, if both uses have the same start time then they will perform identical, and redundant work. We can use lazy memoisation to solve this problem. For each behavior and event, we can create a map which relates start times to *started* behaviors or *started* events. When we start an event we first look up in the map the current start time: if the event has already been started at that time we can just use the stored value. Otherwise we must start it for and store it.

```
memo :: (Time -> IO a) -> (Time -> IO a)
memo f = unsafePerformIO $ do
 mp <- newWeakMap
 return $ \t -> do mb <- get mp t
                   case mb of Just a -> return a
                              Nothing -> do
                                   a <- f t
                                   put mp t a
                                   return a
```

To make this space-efficient we can use a `WeakMap`. This makes use of weak pointers and finalisers to guarantee values that are no longer needed will be cleared out when their keys are no longer accessible.

```
newWeakMap :: IO (WeakMap a)
put :: WeakMap a -> Time -> a -> IO ()
get :: WeakMap a -> Time -> IO (Maybe a)
```

7.4.5. Basic Behavior Combinators

We can define the Behavior combinators in a similar manner to the event combinators. For instance, we can define `stepper` as follows. We create a new behavior which when started, starts its argument event. It then starts listening to changes after that.

```
stepper :: a -> Event a -> Behavior a
stepper a (Event mke) = Behavior $ memo $ \t0 -> do
 e <- mke t0
 ref <- newIORef a
 let alter v = addDelayedAction $ writeIORef ref v
 addL t0 e (mkL alter)
 let sample t = do v <- readIORef ref
                   return (v,True)
 return $ B sample (mkEv $ addL t0 e $ \l -> tellL () l)
```

The definition is very similar to that contained in section 7.2.7.4. It has two important differences. Firstly, it is a pure function as it performs the necessary initialisation actions as part of its "start action". Secondly, we add the alter listener to e so that it hears occurrences strictly after the start time. Again this is done because the Wan-Hudak semantics state that a behavior can only hear change events immediately after its start time.

7.4.6. Implementing Snapshot

For snapshot we might expect to start the event, start the behavior and then make a new event which adds modified listeners to the old event which sample the behavior before firing. However, here we have a big problem: *snapshot has to be lazy in b for self-reactive behaviors to work.*

So we have a workaround. Instead of starting b immediately, we delay starting b by using a delayed action. The delayed action still uses the same start time for b, it just performs the action slightly later. It then writes the started behavior to the `IORef`. The snapshotting event then just reads the behavior from this `IORef` to snapshot it.

There are two potentially critical problems here. Firstly, we would have a serious problem if we tried to snapshot the behavior before the delayed action fired, because the value in the `IORef` would be undefined. Fortunately, the formal semantics saves us from this: events can only hear occurrences *strictly after* their start time. This first problem cannot therefore occur. To prevent it we again use `addL` to guarantee that listeners added to the new event only hear occurrences strictly after the start time.

The second problem comes from adding a delayed action which starts b. If b itself involved a snapshot then this could add another delayed action, that also needs to be run. With a self-reactive definition we might expect to end up in an infinite loop. Fortunately this can be avoided, because all behavior starting actions are memoised. Therefore if we try to start a behavior for a second time with the same start time, we just lookup and use a value. We don't rerun the start code. An infinite loop will therefore be avoided.

```
snapshotE :: Event a -> Behavior b -> Event (a,b)
snapshotE (Event e) b = Event $ memo $ \t0 -> do
  eV <- e t0
  ref <- newIORef undefined
  addDelayedAction $ do {b' <- mkB b t0;
                         writeIORef ref (sampleB b')}
  let sample t = do {sample <- readIORef ref; sample t}
  let fl = liftL1 $ \op (t,v) -> do (b,_) <- sample t
                                    op (t,(v,b))
  return $ \l -> addL t0 eV (fl l)
```

7.4.7. Summary

The implementation outlined in this section appears to be promising. The Wan-Hudak semantics seem to sit well with a data driven implementation. This approach has not, however, been fully tested. Further work is required to determine whether this approach really does provide a robust, efficient, implementation faithful to their FRP formal semantics.

7.5. Implementing Behavioral Collections

7.5.1. Introduction

The implementation of dynamic collections requires some clever coding. To work efficiently it should be possible to render them incrementally and treat them as standard behaviors. To achieve this we can consider a collection to consist of two parts, a simple behavior representing its value at any given time, and an event generating individual incremental changes. There is one important issue that must, however, be considered. This issue is best understood in the context of a concrete example. For instance, imagine if we were to implement a dynamic collection of type `Coll`, with two incremental updates, insert and delete. We might consider representing it as follows.

```
type Coll a
data CollB a = CollB (Behavior (Coll a)) (Event (CollUpd a))
data CollUpd a = Insert a | Delete a
```

This representation has two unfortunate consequences which arise when trying to implement a map function. We could not implement a map function with the following standard type.

```
map :: (a -> b) -> CollB a -> CollB b
```

We would instead have to guarantee that the result type of the argument function also had equality defined upon it. Furthermore, the notions of equality would have to be equivalent. In other words map could only be implemented with *injective* functions.

```
map :: Eq a,Eq b => (a -> b) -> CollB a -> CollB b
```

For instance, if we were to apply `f` to a list of type `a`, for `x` and `y` of type `a`,

```
(x == y) <==> f x == f y
```

This implementation also has an efficiency problem; when mapping a function across the collection update we would end up applying the function to both `Insert` and `Delete` updates, which might end up being fairly expensive.

Instead, we can associate a unique identifier with each element in the collection. We can then refer only to this unique identifier when deleting items. This guarantees that altering the value of each entry will have no effect on `Delete` operations.

```
data CollB a=CollB (Behavior (Coll (Entry a))) (Event (CollUpd a))
data Entry a = Entry {ename :: Ident, eval :: a}
data CollUpd a = Insert Entry | Delete Ident
```

We can generalise this approach, to define a parameterised dynamic collection type. It is used to model a collection of type "`c a`". A value of type "`evop a`" represents an incremental update to a collection. Internally we store a behavior containing a collection of values of type "`Entry a`".

```
data CollectionB evop c a =
 CollectionB (Behavior (c (Entry a))) (Event (evop a))
```

We could, in fact, generalise this still further by parameterising over the `Entry` type. In the case of sets and lists, the two dynamic collection types implemented so far, this representation is sufficient.

```
data CollectionB f evop c a =
 CollectionB (Behavior (c (f a))) (Event (evop a))
```

The next important question becomes, how do we generate these unique values. Section 7.5.2 will show how this can be done with list collections; section 7.5.3, will then generalise this mechanism, discussing briefly how it applies to sets.

7.5.2. Implementing Lists

We can represent a dynamic list as a collection and a list update. List updates either consist of a single reset action, or a list of modifier actions that insert new items at a given position, delete a given item, move an item to another location in the list.

```
type ListB = CollectionB Entry ListUpdates [] a

data ListUpdates a = LUpds [ListUpdate] | ResetL [ListEntry a]

data ListUpdate a = InsertL [ListEntry a] Pos
                  | DeleteL Ident
                  | MoveL Ident Pos
```

```
data Pos = Front | Back | Before Ident | After Ident
```

7.5.2.1. The Static IList Type

The important question is, how do we generate unique identifiers for entries. Section 4.9.1 presented an interface for dynamic lists in which programmers do not have explicit access to the `ListUpdates` type. Instead, recall that a dynamic list is generated using the static incremental list (`IList` type), which supports the standard Edison `Sequence` interface[28].

```
mkListB :: IList a -> Event (IList a -> IList a) -> ListB
```

It is this `IList` type that can be used to generate unique names and updates. We can define the `IList` type as shown below.

```
data IList a = IList [Entry a] NameGenerator (ListUpdates a)

newtype NameGenerator =
  NameGenerator (forall a . a -> (Entry a,NameGenerator))
```

An incremental list consists of four parts, a list of entry elements which represents the current state of the list, a unique name generator and the updates so far. Note that the generator is a data type containing a polymorphic function that will generate an Entry value for any type. This `IList` may have been newly generated, for instance, by using the `empty` constructor. Alternatively, the `IList` may have been generated by simply modifying the previous list. For instance, by inserting a single element. If the `IList` is a new list then the updates will be a `Reset` update. Otherwise it will contain a list of inserts, deletes and moves. We will refer to a list made from a previous one as a *modified list.*

We can now define the basic constructors. For instance, to create an `IList` from a list of values, we generate a list of entries, and a single reset update. This function uses `toEntries`, which is simply a list version of the basic name generator.

```
fromList :: [a] -> IList a
fromList xs =
 case toEntries xs initGen of
   (xs,gen) -> IList xs gen (ResetL xs)

initGen :: NameGenerator
toEntries :: [a] -> NameGenerator -> ([Entry a],NameGenerator)
```

We can insert an element at the beginning of a list, using `cons`. This generates a unique name for the new element, adds it to the list, and adds an insert update to the list of updates.

```
cons :: a -> IList a -> IList a
cons a (IList es gen us ) =
 case gen a of
  (e,gen) -> let es' = (e:es)
             in IList es' gen (addUpds es' [InsertL [a] Front] us)
```

When adding more updates, there are two possibilities: (1) we are dealing with a modified list, in which case we add the new updates to the old updates; (2) we are dealing with a reset. In this case, the new list is not a modified list, so one reset update will be enough to change the list. We change this operation to reset to the latest list value.

```
addUpds :: [Entry a] -> [ListUpdate a] -> ListUpdates a
        -> ListUpdates a
addUpds es us (LUpds us') = LUpd (us++us')
addUpds es _ _ = ResetL es
```

[28] Recall that the Haskell Edison library [145], defines a general interface for dealing with functional data structures such as `Sequences` and `Sets`.

We can define `filter` fairly simply, we partition the list based on the predicate. We keep all the elements that satisfy the predicate (forming the new list), and delete every element that does not satisfy the predicate. Other delete functions would, therefore, use a similar approach. Note that for deletion equality of two entries is not dependent on their actual values. Two entries with the same value will still have different unique identifiers. We can therefore define a variety of delete functions, for instance, the standard *drop* function, which drops the first n elements of a list. This would have been significantly more difficult if deletion equality was dependent on equality between element values, as a list could only safely have one entry with a given value.

```
filter :: (a -> Bool) -> IList a -> IList a
filter p (IList xs n ops ) =
 case L.partition (p . eitem) xs of
  (keep,remove) ->
       IList keep n
             (addUpds keep (map (DeleteL . ename) remove) ops)
```

We can implement reverse in terms of a series of move actions. We reverse the actual entry list, and then generate a series of Move updates, that move the element to the back of the list. As we are accumulating a reversed sequence of updates (we always insert new updates at the beginning of the update list) this will have the effect of reversing the list.

```
reverse :: IList a -> IList a
reverse (IColl xs n ops ) =
  let new = reverse xs
      op = map (\e -> MoveL (ename e) Back) xs
  in IList new n (addUpds new op ops)
```

When appending two lists we need to check if either is a modified list. By doing this we can ensure that we generate an insert update instead of a reset update where possible, as this will be more efficiently, incrementally rendered. Both lists will have been independently using the name generation function. We therefore need to rename one of the lists, to guarantee that both use the name space. We then simply append both entry lists. If neither list is a modified list we generate a reset update, otherwise the new list is added to the modified list. Note that if both are modified lists we need to pick one as the real modified list and one as the new list.

```
append :: IList a -> IList a -> IList a
append l1@(IList ls1 n1 us1 ) l2@(IList ls2 n2 us2 ) =
 if isModified us2 then
  let (ls1',n) = toEntries (map eitem ls1) n2
      new = ls1' ++ ls2
  in IList new n (addUpds new [InsertL ls1' Front] us2)
 else if isModified us1 then
  let (ls2',n) = toEntries (map eitem ls2) n1
      new = ls1 ++ ls2'
  in  IList new n (addUpds new [InsertL ls2' Back] us1) b2
 else
  let (ls1',n) = toEntries (map eitem ls1) n2
      es = ls1' ++ ls2
  in IList es n (ResetL es)
```

We can easily define destructors that access the `IList`, such as `toList`.

```
toList :: IList a -> [a]
toList (IList xs _ _) = map eitem xs
```

We can therefore implement a complete set of functions to access, create and modify incremental lists. When using these functions, it is obviously more efficient to modify the old list than completely replace it.

We now need to define the dynamic list constructor function. We generate an event that accumulates the current `IList`, starting with the initial `IList`. It does this using the apply function. When applying a list update, we are only interested in the current entry list and name generator; we drop the

old list of updates, so that each event occurrence simply generates a new list of updates. To generate the update event, we simply extract the updates list (reversing it, so that updates are now correctly ordered). To generate the behavior, we simply step through every list of entries. Note that when using the data-driven FRP implementation, `scanlE` and `stepper` are IO actions, and so `mkListB` must also become an IO action; if using the hybrid implementation, `mkListB` could be a pure function.

```
mkListB :: IList a -> Event (IList a -> IList a) -> IO (ListB a)
mkListB l e = do
  (ev' :: Event (IList a)) <- scanlE apply l e
  let opsev :: Event (ListUpdates a)
      opsev = ev' ==> toOps
  (b :: Behavior [Entry a]) <- stepper (toList l) (ev' ==> toList)
  return (CollectionB b opsev)

 where apply :: (IList a -> IList a) -> IList a -> IList a
       apply f (IList ls n _) = f (IList ls n (LUpds []))

       toList :: IList a -> [Entry a]
       toList (IList ls _ _) = ls

       toOps :: IList a -> [ListUpdate a]
       toOps (IList _ _ u@(ResetL _)) = u
       toOps (IList _ _ (LUpds us)) = reverse us
```

We therefore have a simple, elegant implementation of the dynamic collection constructor, which allows dynamic lists to be created using a high-level mechanism, and is efficient to use.

7.5.2.2. ListB operators

We now need to define the dynamic list combinators themselves. These fall into three general categories. Good examples of these categories are provided by `map`, `filter` and `filterB`.

The simplest class of combinators are those that only need to apply a given function to the elements in the dynamic list, such as map. To map a function across a list, we apply the function to each entry in the behavior, and to the value contained in each update (clearly only when it is an `Insert` update do we need to do anything.) The `fmap` function is implemented by any member of the `Functor` class, and supports mapping a function across a data type.

```
map :: (a -> b) -> ListB a -> ListB b
map f (CollectionB b e) = CollectionB (lift1 (fmap (fmap f)) b)
                                      (e ==> fmap f)
```

The second class of functions are those that apply static functions to a dynamic list. These include the basic filter function, which applies a static filter function to the list. This is used to filter the dynamic list for all time.

We can implement this as follows.

```
filter :: (a -> Bool) -> ListB a -> ListB a
filter p (CollectionB b e) =
  CollectionB (lift1 (filter (p . eitem)) b)
              ((ev `snapshot` b) `mapMaybeE` handle)
```

To modify the behavior, we can simply filter it. To modify the update event, we need to snapshot the previous list and apply the handle function to the pair of values. If the update is a reset, we simply filter the new reset list. If not, we need to apply each of the individual updates.

```
    where
     handle :: (ListUpdates a,[Entry a]) -> Maybe (ListUpdates a)
     handle (ResetL r,_) = Just (ResetL (filter (p .eitem) as))
     handle (LUpds us,es) = LUpds (snd (mapAccumMbL doOne es us))
```

Each of the update functions refers to a list position, which is valid in the old list, but may not be valid in the new list. We therefore need to translate between old positions and new positions. To do this we need access to the original list. We have access to, `es`, the snapshot result, which represents the list state just before the current updates. As we test each update, we therefore need to accumulate a new list, by applying each individual update in turn. We therefore use `mapAccumMbL`, which maps an accumulating function along a list. We use `applyLUpd`, at each step to apply the latest update to the entry list, to generate a new entry list.

```
mapAccumMbL :: (a -> b -> (a, Maybe c)) -> a -> [b] -> (a, [c])
```

When applying an insert update, we check whether the item should be in the list. Only in this case do we pass on the insert action. We translate the location to a new location using `realLoc`. When deleting an item from a list, we lookup the item, and again only delete it if the item satisfies the predicate (and is therefore in the filtered list). On a move update, we again check if the item satisfies the predicate. Only then do we move it to its new recalculated position.

```
    doOne :: [Entry a] -> ListUpdate a
          -> ([Entry a],Maybe (ListUpdate a))
    doOne (op@(InsertL ent@(Entry id a) pos),bs) | p a =
       let upd = (InsertList n a (realLoc p bs pos))
        in (applyLUpd op es,Just upd)
    doOne (op@(DeleteL n),bs) | maybe False p (lookup n bs) =
      Just (applyLUpd op es,Just op)
    doOne (MoveL x pos,bs) | maybe False p (lookup x bs) =
          let upd = MoveL x (realLoc p bs pos)
          in (applyLUpd op es,Just upd)
    doOne es op = (applyLUpd op es,Nothing)

applyLUpd :: ListUpdate a -> [Entry a] -> [Entry a]
realLoc :: (a -> Bool) -> Entry a -> Pos -> Pos
```

Both of the above classes of function apply their updates to behavior and update events separately. In theory, this might result in a performance problem; however, in practice it does not appear to be a major issue. There are two cases when we may end up with a problem; (1) if we were to apply such an approach when mapping an IO action across a dynamic list. In this case we would end up applying the action more than once, which would be incorrect; (2) when applying behavior based functions to update a collection, such as `filterB`, where we end up with unacceptable performance.

We can implement `mapIOLB`, by applying the action to the event, and then generating a new list. We must sample the behavior at the current time, to get the initial state for the new dynamic list.

```
mapIOLB :: (a -> IO b) -> ListB a -> IO (ListB b)
mapIOLB f (CollectionB b e) = do
 t <- getTime
 curr <- b `at` t
 w <- newWire
 addListener e (mapIOL f (input w))
 curr <- mapIO f curr
 b <- stepper curr (event w)
 return (CollectionB b e)
```

The implementation of functions such as `filterB` requires some sophistication. Recall that these combinators each take a function to extract a behavior from a list element, and a function valued behavior to apply. We need this flexibility, because the elements of a dynamic list may themselves be dynamic.

```
filterB :: (a -> Behavior b) -> Behavior (b -> Bool)
        -> ListB a -> ListB a
```

We need to generate an update in one of three cases: (1) when the initial dynamic list generates an update; (2) when the value of the predicate behavior changes; (3) when the value of one of the extracted behaviors changes.

To achieve this we create an accumulating event, containing the current Boolean filter function; a list of elements, one for each dynamic list entry, with an identifier, list value, current extracted behavior value, and a Boolean representing the result of applying the filter function to the extracted behavior value; and finally, the last list-update operations. Every time the filter function changes, a list operation occurs, or a value changes, this event should generate an occurrence.

```
type ListItem a b = Entry (a,b,Bool)
type AccumInfo a b = (a -> Bool,[ListItem a b],[ListUpdates a])
```

To generate the change event that updates this filter function we need to first make a behavior list with an entry containing not only the value a, but the behavior b and an event stream of type b, that occurs every time the behavior b changes. This assumes the existence of a function, `toStream` which turns a behavior into an event stream. Such a function could be implemented using `newWire`, and the workpool interface discussed in Section 7.2.7.6.

```
fromA :: (a -> Behavior b) -> a -> IO (a,Behavior b,Event b)
fromA fb a = do
   let b = fb a
   e <- toStream b
   return (a, b, e)

toStream :: Behavior a -> IO (Event a)
```

We apply this function to the list. We also extract an event for the predicate, which generates an occurrence every time the value of the predicate behavior changes.

```
filterB extract predB l = unsafePerformIO $ do
  l@(CollectionB beh1 ev1) <- mapIOLB (fromA fb) l
  predE <- toStream predB
```

We then sample the current state of the new list. For each item in the list, we sample the extracted behavior, and compute whether the item should be in the new list or not. We use this to generate an initial `ListItem` for each entry and an initial list of items that will be in the new dynamic list.

```
  let check :: Entry (a,Behavior b,Event b)
            -> IO (Maybe (Entry a),ListItem a b)
      check (Entry id,(a,b,_)) = do
       v <- b `at` t
       if pred v then return (Just (Entry id a),
                              Entry id (a, v, True))
                 else  return (Nothing,
                               Entry id (a, v, False))

  t <- getTime         -- get the time
  pred <- at predB t   -- sample the predicate at that time
  xs <- at beh1 t      -- sample the initial list at that time

  (inits,gens) <- fmap unzip $ mapM check xs
```

We then generate the accumulating info event and finally make a new list from its updates and the initial list of values.

```
  co <- changeOpsE xs ev1
  e <- (accumE (pred, gens, []) $
               predE ==> flip changePredOp
               .|.
               co ==> flip changePredEvs)

  return $ newListB (catMaybes inits) $ e ==> threeVal
```

The changeOpsE function generates a change event. This consists of either a list update, with a value of type (a,b) or a change event marking an alteration in one of the updated behaviors. We do this by using the event stream from the list item, and allowing it to generate occurrences until its associated entry is deleted.

```
data Change a b = ValChange Ident b | ListOp (ListUpdates (a,b))

changeOpsE :: [Entry (a,Behavior b,Event b)]
           -> Event (ListUpdates (a,Behavior b,Event b))
           -> IO (Event (Change a b))
```

We update the list information using the change event, creating an insert update when an element becomes visible, and a delete update when an element becomes invisible. When a value changes, we generate an insert or delete update as appropriate. When the predicate event changes, we apply the new predicate, and alter only those items whose values have changed.

```
changePredOp :: ListInfo a b -> (b -> Bool) -> ListInfo a b

changePredEvs :: ListInfo a b -> Change a b -> ListInfo a b
```

This mechanism is complex but reasonably efficient. It also generalises to cover the complete class of functions that accept behavioral updates. For instance, to perform a behavioral sort we use a similar mechanism, except that we accumulate the ordering function, instead of a predicate.

7.5.3. Generalising the collection approach

We can generalise much of the approach in the previous section to allow it to apply to any form of collection. If we parameterise over the individual update and collection type, we can reuse the list update, and incremental list type.

```
data CollUpdates upd c a = Reset (c a) | CUpds [upd a]
data IColl upd c a =
 IColl (c (Entry a)) NameGenerator (CollUpd upd c a)
```

We can define the list equivalents of these as follows.

```
type ListUpdates a = CollUpdates ListUpdate [] a
type IList a = IColl ListUpdate [] a
```

Using this mechanism, we can define the set equivalents. A set update, consists simply of an insert and delete update, but no move operation. We could easily imagine applying a similar mechanism to other types of collection, such as trees.

```
type ISet a = ISet SetUpdate Set a
type SetUpdates a = CollUpdates SetUpdate [] a
data SetUpdate a = InsertS (Entry a) | DeleteS Ident
```

Given this type, we can define a general dynamic collection constructor that creates a general collection out of an incremental static collection and event.

```
mkCollB :: IColl upd c a -> Event (IColl upd c a -> IColl upd c a)
        -> CollectionB Entry (CollUpdates upd) c a
mkCollB l e = do

  (ev' :: Event (IColl upd c a)) <- scanlE apply l e
  let opsev :: Event  (CollUpdates upd c a)
      opsev = ev' ==> toOps

  (b :: Behavior [Entry a]) <- stepper (toColl l) (ev' ==> toColl)
  return (CollectionB b opsev)
```

```
where apply :: (IColl upd c a -> IColl upd c a)
            -> (IColl upd c a -> IColl upd c a)
      apply f (IColl ls n _) = f (IColl ls n (CUpds []))

      toColl :: IColl upd c a -> c (Entry a)
      toColl (IColl ls _ _) = ls

      toOps :: IColl upd c a -> CollUpdates upd c a
      toOps (IColl _ _ u@(Reset _)) = u
      toOps (IColl _ _ (CUpds us)) = reverse us
```

We can also reuse similar mechanisms when implementing the remaining dynamic collection combinators. For instance, we can define a map function for any dynamic collection, whose static collection type support mapping (i.e. is an instance of the `Functor` class). We therefore have a generic dynamic collection type that should be usable with any functional data structure.

7.6. Summary

This Chapter has presented the important issues in the implementation of the FranTk core. The most difficult was the development of an efficient, correct, robust implementation of the core FRP combinators. This Chapter discussed the two current semantic models for FRP, the first by Elliott and Hudak; the second by Wan and Hudak. This chapter discussed three possible implementations. The first is a purely data-driven implementation. This implementation is efficient and robust. Unfortunately, it requires a change in the type of any combinator that relies on an event's history. The data-driven representation does not store an event's history and therefore any such combinator must be an IO action. In FranTk, this is not a particularly problematic restriction, as all FranTk programs use the GUI monad. The second implementation is a hybrid that combines the streams and data-driven approaches. It attempts to remain faithful to the Elliott-Hudak FRP semantics. Unfortunately, this implementation is not entirely robust. In particular, the use of `merge` is not referentially transparent in the presence of simultaneous event occurrences. This implementation therefore serves more as an example of the difficulties that can arise when implementing FRP. This implementation can be more prone to space leaks as it keeps track of an event's history. The final implementation discussed in this Chapter is a refinement of the data-driven implementation that attempts to satisfy the Wan-Hudak semantics. This approach appears promising. However, further work is required to test and validate it. The development of a truly efficient, robust implementation that is faithful to the formal semantics of FRP remains a topic for future research.

Chapter 8 - Toolkit independence in FranTk

The implementation of FranTk consists of two parts; (1) implementation of the core FRP combinators and dynamic collections; (2) implementation of the FranTk widget toolkit. The previous Chapter covered the former, the latter will be discussed in this Chapter.

Though FranTk has been implemented on top of Tcl-Tk, the widget set has been implemented in as toolkit independent a manner as possible. This should make it relatively easy to port to other toolkits[29]. To achieve this the toolkit implementation consists of four parts.

1. Initial Exports – this interface should define some basic primitive types used throughout FranTk.
2. Toolkit independent widgets – these define an abstract widget classes interface, which provides an imperative widget interface, and the GUI monad.
3. Components interface – this defines behavioral widgets, and components in terms of the abstract widget interface.
4. Toolkit implementation – this provides a toolkit specific implementation of the abstract widget interface, such as a Tcl-Tk interface, and provides a set of widget components that can be created.

8.1.1. The Abstract Widget Interface

The abstract widget interface defines four different classes, that should be implemented by a toolkit.

1. Widget Item – the widget item interface should be supported by any widget. It supports four actions. A widget can be destroyed; it has a unique identifier; it can be configured to set values such as colour, using a `Config` type which is exported by the initial exports interface; it has an `accepts` function which specifies whether the widget accepts a given configuration option; finally, it has a method to add a finaliser, which is an action that will be run when the widget is destroyed.

```
class WidgetItem w where
 cset :: w -> [Config] -> IO ()
 accepts :: ConfigName -> Bool
 destroy :: w -> IO ()
 uniqueId :: w -> Ident
 addFinaliserW :: w -> IO () -> IO ()
```

2. Window Item – a window item is a top-level window. It can be mapped, unmapped, iconified, and deiconified. We can also get a panel from it to display panel item widgets in. It is assumed that there will be one window panel in the window. We should make sub panels from the window panel the rest of the time. The idea of using panel's here is similar to the Java Swing toolkit.

```
class WidgetItem w => WindowItem w where
 showWindow :: w -> IO ()
 hideWindow :: w -> IO ()
 iconifyWindow :: w -> IO ()
 deiconifyWindow :: w -> IO ()
 getWinPanel :: w -> IO Panel
```

3. A Panel item – A panel item is a widget that can be added to a panel. In particular, it can be added to a grid or box, formed from the panel. Box items have an ordering, and a panel item can be raised and lowered within that ordering.

```
class WidgetItem w => PanelItem w where
 gridAdd :: w -> Grid -> GridLoc->[GridBagConstraint]-> IO Remover
 boxAdd :: w -> Box -> PlacePos Ident -> [PackInfo] -> IO Remover
 raise :: w -> Maybe Ident -> IO ()
 lower :: w -> Maybe Ident -> IO ()
```

[29] For instance, there are currently plans to port FranTk to use the GTK Open Source GUI Toolkit. Information on GTK can be found at http://www.gtk.org.

4. A Canvas item – a canvas item, is a graphics item that can be placed on a graphics canvas. Items added to such a canvas are placed at a specific location. This location is defined by a transformation; the `transformItem` function transforms a canvas item in terms of the canvas co-ordinates. A canvas item can be mapped, unmapped, and raised or lowered in the canvas stacking order.

```
class WidgetItem w => CanvasItem w where
 showItem :: w -> IO ()
 hideItem :: w -> IO ()
 transformItem :: Transform2 -> w -> IO ()
 raiseItem :: w -> Maybe Ident -> IO ()
 lowerItem :: w -> Maybe Ident -> IO ()
```

A panel is a data type that supports three functions, to create a subpanel, make a grid from the panel, and make a box from the panel.

```
data Panel = Panel {mkSubPanel :: IO Panel,
                    setGridLayout :: IO Grid,
                    setBoxLayout :: IO Box}
```

Grids allows elements to be laid out in a 2-D grid. Boxes allow items to be laid out in either a horizontal or vertical form. Items can also be padded, or made to fill extra space. Both of these layout are very general and are supported, for instance, by both Tcl-Tk and Java layout managers.

Any widget that supports user input, should implement the `Bindable` interface. The `bind` function adds a listener to the widget, for a particular action. The `Action` type specifies which type of input we are interested in (such as mouse clicks). The listener hears general values of type `UserAction`. This returns a remove action to unregister interest in the widget.

```
class WidgetItem w => Bindable w where
 bind :: w -> Action -> Listener UserAction -> IO (IO ())
```

The abstract widget interface also defines the `GUI` monad. This monad passes around a value of type `GUIDef`, which contains a number of values. The first of these is the explicit state for accessing the underlying toolkit, which is a value of type `GUIInfo`, imported as one of the initial exports. It also contains a variable for a unique name supply, a mutable set of remove actions for active widgets, a reference to the current, and root window, and to the current canvas, if there is one.

When defining the GUI monad we need to lift a reasonable set of IO actions into the GUI monad. In addition, we lift the Listener implementation into the GUI monad. To do this we define a new listener type, that is a function from a `GUIDef` value representing the state of the GUI monad to primitive listeners. We can therefore define GUI action listeners using this mechanism.

```
type Listener a = GUIDef -> Prim.Listener a
```

We lift the basic listener operations so that they work on this new type. For instance, we can redefine `liftL1` as follows. Recall from Section 7.2.2 that the whole of the listener algebra is defined in terms of three of these basic lifting functions.

```
liftL1 :: (Listener' a -> Listener' b) -> Listener a -> Listener b
liftL1 f l = \g -> L.liftL1 f (l g)
```

We lift the `addListener` function so that it too works within the GUI monad. This provides the listener with the current `GUIDef`. The listener's action is therefore evaluated in the environment context that it was added in.

```
addListener :: Event a -> Listener a -> GUI (GUI ())
addListener e l = do
 st <- getGuiDef
 rm <- liftIO $ Prim.addListener e (l st)
 return (liftIO rm)
```

Every widget in FranTk is wrapped in a `PrimWidget`. A `PrimWidget` has access to the `GUIDef` state from when it was create, the widget itself, an action to hide the widget, and a weak reference to the hide action. It also has a variable storing the names of all the `Config` options that have already been applied to the widget. This is used when displaying a *Widget Behavior* (see Section 8.1.2). .

```
data PrimWidget w = PrimWidget {
    guiDef :: GUIDef,
    widgetPrim :: w,
    hideRef :: RemRef,
    hideRefWP :: (Weak RemRef),
    usedconfigs :: IORef [ConfigName]
    }
```

To make a `PrimWidget`, we first extract the current `GUIDef`. We then create the widget using the current window. We make a variable to hold the hide action for the widget. We then create a weak reference to this hide variable, with a finaliser that destroys the widget. Recall that in `GUIDef` there is a mutable list of visible items. When a widget is visible, then its hide variable goes in this list. This variable remains live until the widget becomes invisible and there are no other references to it that could make it visible again. At this point we can safely destroy the widget using the finaliser. When we show the widget we make it active by adding its remover variable to the active widget set. We then place the remove action for the set in the remover `IORef`. When we make the widget inactive we can then just run the remover `IORef`.

We can now define typed configuration options. A configuration option is an action that can be applied to a widget. The action has perhaps a configuration name associated with it, if the action is associated with a given configuration option. A configuration option therefore has a notion of equality. Two configuration options are equal if they have configuration names that are equal.

```
data Conf w = ConfGUI (Maybe ConfigName) (w -> GUI ())
```

We can define behavioral configuration options in terms of static ones, using the workpool interface. Given a `Config` name, a function to create a `Config` from a value, and a Behavior; we create a configuration option. We add a listener to the behavior that sets the configuration option every time the value changes. We add the remove action as a finaliser for the widget. This guarantees that the widget will be updated after it is destroyed.

```
confB :: ConfigName -> (a -> Config) -> Behavior a ->Conf w
confB confname conf val =
 ConfGUI (Just confname)
        (\w -> do rm <- addListener val (mkL $ cset w . conf)
                  addFinaliserW w rm)
```

One useful optimisation is to only apply configuration updates when a widget is actually visible. When invisible we record any changes that must be made to the widget, by storing the latest update action in a finite map, with an entry for each `ConfigName`. When the widget is redisplayed we then simply apply all the latest updates. This approach is particularly useful if some aspect of a widget's appearance is updated regularly when it is invisible.

8.1.2. Components and Widget Behaviors

Widget behaviors are defined in terms of an abstract widget type. They may be dynamic and map down onto several primitive widgets. We define a generic parameterised abstract widget type, and then several specific instantiations of it: for a top-level window; standard widgets, such as buttons and labels; and canvas items, that represent graphical objects. Note that each of these map down to a specific interface type, defined in the abstract widget interface. This approach was inspired by the Fran ImageB type, which provides an abstract data representation of a dynamic image.

A generic widget is an abstract data type. It may be a simple widget, an empty widget, the composition of two widgets, a dynamic pile of widgets (modelled as a list behavior), a behavioral conditional choice or a widget switcher. We may add a listener to all of the input in a widget behavior, using `GrabInput`. Finally, we may associate a list of style configuration options with a widget.

```
data GWidgetB a =
       GWidgetB  a
     | EmptyW
     | GrabInput  Action  (Listener UserAction)  (GWidgetB a)
     | Compose  (GWidgetB a)  (GWidgetB a)
     | PileW  (ListB (GWidgetB a))
     | CondW  BoolB  (GWidgetB a)  (GWidgetB a)
     | SwitcherW  (GWidgetB a)  (Event (GWidgetB a))
     | WithStyle [Conf Style] (GWidget a)
```

We also define a generic component type based on the generic widget type.

```
type GComponent a = GUI (GWidgetB a)
```

We define specific instantiations of this widget type. For instance a `WindowWidgetB` is a generic widget containing a window item. Here we use an existential type to wrap a primitive widget. A window widget can therefore contain any object that is an instance of the `WindowItem`, and `Bindable` classes.

```
type WindowWidgetB = GWidgetB WW
data WW = forall w . (Bindable w, WindowItem w)=>WW (PrimWidget w)
```

Standard widgets are based on the `PanelItem` interface. They may also specify layout information. This will either be a packing mode (above or beside), or new pack information (either static or behavioral), specifying whether widgets are to be padded, or expanded when placed in a box. We can use a similar mechanism to place objects in a grid.

```
type WidgetB = GWidgetB PW
data PW = forall w . (Bindable w ,PanelItem w) =>PW (PrimWidget w)
        | WithPack [PackInfo] WidgetB
        | WithPackB (Behavior [PackInfo]) WidgetB
        | WithPackMode PackMode WidgetB
```

We can define a canvas widget in a similar manner, where items are behavioral transformations, or primitive widgets based on the `CanvasItem` class.

```
type CanvasWidgetB = GWidgetB CW
data CW = forall w . (CanvasItem w,Bindable w) =>CW (PrimWidget w)
        | Transform2W Transform2B CanvasWidgetB
```

We can define the display combinators in terms of these data types. For instance, the above combinator sets the packing mode to `PackAbove`, and then composes its two argument widgets.

```
instance Packable WidgetB where
 above w1 w2 = GWidgetB $ WithPackMode PackAbove $ Compose w1 w2
```

The `nabove` combinator (which packs a dynamic list of widgets above each other) simply sets the pack mode to `PackAbove`, and then generates a pile from the given dynamic list.

```
instance PackCollection ListB WidgetB where
 nabove ls = GWidgetB $ WithPackMode PackAbove $ PileW ls
```

We can easily define the `Component` interface on top of this widget interface. For instance, we define the above combinator as follows. It simply runs both of its child widget actions, and then places the two widget behaviors above each other.

```
instance Packable Component where
 above w1 w2 = do x <- w1
                  y <- w2
                  return $ x `above` y
```

To place a dynamic list of Components above each other, map across the ListB, applying each GUI action, to generate a ListB of widget behaviors. We then just compose this new list.

```
instance PackCollection ListB Component where
 nabove ls = do ws <- mapGUI id ls
                return $ nabove ws
```

Given these abstract types, we define a set of display functions, that display window, standard and canvas widget behaviors. We can define a display function, that displays an abstract widget. This requires a function to display a primitive item; a function to make a group for moving items in the stacking order; some type specific display information, and a value of type DisplayInfo which represents the current display state.

```
displayW :: (DisplayInfo -> b -> a -> IO ())
         -> (DisplayInfo -> b -> IO (MkMoveOp b))
         -> DisplayInfo -> b
         -> WidgetB a
         -> IO ()
```

We use the DisplayInfo type to pass information down to all child widgets. Widgets may be visible at any given time; will be deleted on a given event; appear in some form of pile; may have listeners on their user input; and may require access to a unique name supply. There is also a current list of configuration operations, representing the current style.

```
data DisplayInfo = DisplayInfo {visible :: BoolB,
                                dieE :: (Event Bool),
                                pArray :: PileArray,
                                bindEvents :: [Bind],
                                nameRef :: IORef Int,
                                currentStyle :: [Conf Style]}

type Bind = (Action,Listener UserAction)
```

The PileArray type models a mutable *Pile* of objects. Objects can be made visible and invisible and can be moved about in the stacking order. Piles can also have *Groups*. A group models a group of elements that can be moved en-masse. Groups can be moved in relation to other groups, or to other pile elements. The PileArray type is implemented using an extensible, mutable array, for fast update and access. Each element has a unique abstract identifier. Each entry also has a value of type Ident, which represents the name of the object it is referring to (such as the widget Id). Elements have a Boolean saying whether visible, and a pointer to the element above, below, and to the visible element above and the visible element below.

The implementation of most of the display functions are fairly simple.

- To display an empty widget, we do nothing.
- To display two composed widgets, we display the first, then the second.
- To display a simple widget, we invoke the given display function.
- When we find a new WithStyle definition, we override the current style with the new style information (we can do this because configuration options can be compared for equality).
- When we find a GrabInput we add the action and listener to the bind list. If the action is already in the bind list, we simply merge the new listener with the current listener.
- When displaying a conditional widget, we render both widgets, modifying their visibility with the conditional Boolean behavior.
- When displaying a switcher widget, we first display the initial widget. We add a listener so that on every occurrence, we display the new widget. We stop doing this when the die event from the parent generates an occurrence. When displaying each child widget, we modify its die event, so that it is deleted on the next occurrence from the switching event.

Rendering a dynamic pile of widgets requires a bit more sophistication, and is therefore worthy of further comment.

To render a pile we need support for restacking. To do this we use the grouping function. This first makes a generic group, and returns an operation that can be used by individual stacking groups. This is done by returning a value of type `MkMoveOp`. This contains updated display information, and a generation function. We make specific groups, using this function. To do this we require to provide a name, and an initial position. This action returns a new display information object, and a listener that should be told about restacking of the group.

```
type MkMoveOp b = (DisplayInfo,b,MkMoveOp' b)

type MkMoveOp' b = DisplayInfo -> b -> Ident -> PlacePos Ident
                -> IO (MoveOp,DisplayInfo,b)
type MoveOp = Listener (PlacePos Ident)
```

Given this grouping interface, we render a pile in the following manner. We firstly group the whole pile, by running the function to generate the `MkMoveOp` value. We then sample the current state of the pile (from the list behavior's sampler). We display this initial list, and add a listener to the update event which updates the display. We stop listening to display updates, when the parent's die event generates an occurrence. To display an individual item, we make a group for the item, getting back a move listener. We add the move listener, to the update event, to allow it to hear about restack events. We display the item, altering its die event. This allows us to arrange that the item is deleted either when its parent deletes the whole pile, or when the update generates a delete for the item (i.e. on a reset, or a delete of that element).

The mechanism to generate a group depends on the type of widget we are displaying. When displaying a panel item, we can simply create a new panel, to contain the whole group. On each move update we first move the item in the pile array. We can then restack the panel, using the pile array to compute its new position.

For widget types that do not support such a mechanism, such as canvas widgets, we can use the `PileArray` group support. To use this, we need a lookup table to map from object names (of type `Ident`) to group references, because `ListB` move updates are specified in terms of the entry name, but `PileArray` restacking is specified in terms of group references. We add the object name and group reference to the lookup table at the start, and remove it when the item is deleted. On each move update, we then simply get the group reference of the new object, and move the group. To complete the implementation we pass an event to each primitive widget, that specifies when to restack. The primitive widget then looks up its new position in the pile array, and restacks to the new position.

```
mkMover :: PileArray -- the pile
        -> IORef (FiniteMap Ident GroupRef)
        -> Ident -- the identifier of this object
        -> GroupRef -- the restacking identifier of this object
        -> Event () -- the die event, saying when to stop
        -> IO (Listener' (PlacePos Ident))
mkMover pa fmv ident gref dieE = do
 let findElt v n = do fm <- readIORef v;return (lookupFM fm n)
 updIORef fmv $ \fm -> addToFM fm ident elt
 addListener (onceE dieE) $
   mkL_ (updIORef fmv (\fm -> delFromFM fm ident))
 let moveInPile pos = do
           posg <- mapIO (findElt fmv) pos
           movePileElt pa gref posg
 return (moveInPile)
```

To display an individual primitive widget, we insert it into the pile, associating with the item any groups that it may be in. This generates a unique pile array name for the element. When we show and hide the widget we must update the pile array. When we delete the widget, we delete it from the pile. The use of the pile array is therefore very significant. It keeps track of the position and visibility of each individual item in a particular display area. This ensures that when we display or restack an item, we can find its location in the stacking order of only the currently visible items.

```
insertPileElt :: PileArray -> Ident -> EltName -> [GroupRef]
              -> IO EltName
hidePileElt :: PileArray -> EltName -> IO ()
showPileElt :: PileArray -> EltName -> IO ()
deletePileElt :: PileArray -> EltName -> IO ()
```

We also apply the appropriate display actions for the widget type. We add a listener to the visibility behavior, that shows and hides the item. For instance, we would insert a `Panel Item` widget into the current box, using the current packing mode and options. When the item is hidden, we remove it from the box. We bind all of the current user input listeners to the widget. To apply style attributes, we first lookup which configuration options have already been applied to the widget (using the variable from the `PrimWidget`). These will have been applied when the widget was created, and override any from the style attribute list. We delete any configuration options that have already been used from the current style attribute list. We check which of the remaining style attributes are applicable (using the `accepts` function) and apply these.

8.1.3. Toolkit Dependent Interface

The toolkit dependent interface is then defined on top of this abstract interface. To do this we must define an instance of the relevant classes for each widget. For instance, for a button widget, we must define, an instance, of `WidgetItem`, `PanelItem` and `Bindable`. We also define the available configuration classes using the `Conf` type, with appropriate instances for each available widget. Finally, we define the set of available widget types, with a constructor for each to generate a component.

8.1.4. Implementing Dynamic Documents

The implementation of FranTk also requires an implementation of the dynamic document type. We can implement a dynamic document in terms of some mutable document representation, and an event generating document updates.

```
data DocumentB = DocB {mutDoc :: MutDoc,updDoc :: Event DocUpd}
```

Updates in FranTk, are defined in terms of the structured document type. We can insert a structured update, reset a document to contain a new structured type, or replace one structured update (of a given name) with another. Each of these updates contains a unique identifier, which represents the originator of the update. This may either be the name of a given edit widget, or `noEdit`, if the update came from the application code. This identifier is useful when an edit widget is both sending updates to, and receiving updates from, a dynamic document. In this case an edit widget, will hear the updates it generated. It would clearly be incorrect for it to apply these. The widget can fix this by filtering out all those updates that originated from it (i.e. have its identifier).

```
data DocUpd  = InsertStructured Ident Structured TIndex
             | ReplaceStructured Ident Structured Ident
             | ResetStructured Ident Structured
             | DeleteBetween Ident TIndex TIndex
noEdit :: Ident
data Structured = SText String
                | STag EditTag
                | SGroup [Structured]
                | SNamed Structured Ident
                | STextTagged Structured EditTag
                | SMark EditMark
```

We define a dynamic document in terms of an `IDoc` type. One simple representation of an `IDoc` is a list of document updates.

```
data IDoc = [DocUpd]
```

The `structured IDoc` constructor therefore simply produces a reset update.

```
structured :: Structured -> IDoc
structured = [ResetStructured noEdit s]
```

The `insertStructured` update simply adds an insert update to the current update list.

```
insertStructured :: Structured -> TIndex -> IDoc -> IDoc
insertStructured s t ds = (InsertStructured noEdit s t):ds
```

We generate a document by first accumulating a current `IDoc` update. On every occurrence, we generate a new update list, ignoring the previous update list. Note that as we accumulate the update list in reverse order, we must reverse the order of the updates before passing them on. We then generate a mutable document based on this event.

```
mkDocumentB :: IDoc -> Event (IDoc -> IDoc) -> IO DocumentB
mkDocumentB init e = do
  e' <- accumE init (e ==> \f _ -> reverse (f []))
  md <- mkMutDoc init e'
  return (DocB md e')
```

We now reach the thorny issue of how to implement the `MutDoc`. The preferred representation would be a simple behavior of some static document type.

```
type MutDoc = Behavior Doc
```

We might have a function to generate an empty document, and a function to apply a list of document updates to a document. Using this interface we could generate a `MutDoc` simply using `stepper` and `scanlE`.

```
emptyDoc :: Doc
applyIDoc :: Doc -> IDoc -> Doc

mkMutDoc :: IDoc -> Event IDoc -> IO MutDoc
mkMutDoc id e = do
   let initdoc = applyIDoc emptyDoc
   e <- scanlE applyIDoc initdoc e'
   stepper initdoc e
```

We might then have observation functions to, for instance, extract the text in a document. Using such a static extraction function, we could define the behavioral function `getText`.

```
getTextDoc :: Doc -> (TIndex,TIndex) -> String

getText :: DocumentB -> Behavior (TIndex,TIndex)
        -> Behavior String
getText (DocumentB mutb _) bh = lift2 getTextDoc mutb bh
```

Such a representation relies on an efficient, purely functional, implementation of a document, which has, unfortunately, proved difficult to achieve. Documents need to support insertion and deletion at any given location, which is difficult (though not impossible) to efficiently achieve in a purely functional implementation. More significantly, documents must also support `EditTags` and `EditMarks`. These each take individual configuration information, which may be behavioral, and most significantly may change their locations. We therefore need to convert these index changes into changes in the document. In addition, the `DocumentB` type must support indexing via the `TIndex` type. This allows locations in a document to be referred to in terms of tag and mark indices. These will clearly also change as the document is modified.

One possible representation might be to define a document as a list of lists of segments. A segment represents either a tag start, tag end, a mark or a piece of text. A list of segments represents the contents of one line. A list of list of segments therefore represents the complete document. We maintain a record of the current marks and tags, recording their current location, and relevant configuration information. Unfortunately, such a representation proved too slow when dealing with very large documents. The

development of a truly efficient, functional representation of a FranTk document therefore remains something for future research.

```
data Doc = Doc Tags Marks TextState
type TextState = [[Segment]]
data Segment = TagStart Ident | TagEnd Ident | Mark Ident
             | Text String

type Marks = FiniteMap Ident MarkInfo
data MarkInfo = MarkInfo Ident (Int,Int) [Conf EditMark]
type Tags = FiniteMap Ident TagInfo
data TagInfo = TagInfo Ident (Int,Int) (Int,Int) [Conf EditTag]
```

One solution is instead to really represent a `MutDoc` as a mutable data structure. This restricts us to use of documents only in the IO monad. Observation functions would therefore have to be IO actions.

```
data MutDoc
mkMutDoc :: IDoc -> IO MutDoc
applyIDoc :: MutDoc -> IDoc -> IO ()
getTextMutDoc :: MutDoc -> (TIndex,TIndex) -> IO String
```

As we have moved to the IO monad, a `MutDoc` could either be implemented in Haskell in terms of mutable variables, or in terms of a C data structure. In fact the current FranTk implementation maps the `MutDoc` type down to the C data structure that Tcl-Tk uses to store the contents of an edit widget. This data structure has already undergone a considerable amount of optimisation, and therefore seemed a reasonable choice. This choice obviously makes the `MutDoc` representation Tcl-Tk specific. However, the `MutDoc` type must map exactly to the behavior of an edit widget to allow correct use, so this is a necessary (if unpleasant) restriction anyway.

Given a mutable document representation, we can generate a dynamic document as follows. We generate an initial document from the `IDoc` type. We then add a listener to the event, that updates the mutable document on every occurrence. Note that this uses a delayed action, to guarantee that changes to the behavior happen immediately after the event.

```
mkMutDoc idoc e = do
 md <- mkMutDoc idoc
 let upd ref v = addDelayedAction $ appyIDoc ref v
 addWeakListener e' upd md
 return md
```

If we are using the data-driven behavior implementation, perhaps surprisingly, we can still implement behavior based functions that access the mutable document. For instance, we can implement, `getText` as shown below. Recall that a Behavior consists of an imperative sampler, and an invalidation event. The data-driven implementation already has the pre-condition that it is only safe to sample a reactive behavior at the current time. We can therefore simply implement a sampler that performs an imperative request action on the mutable document. This demonstrates a very powerful feature of the data-driven representation, it allows the implementation of a declarative interface to mutable objects.

```
getText :: DocumentB -> Behavior (Tindex,Tindex)
        -> Behavior String
getText (DocumentB mutd ev) bh =
  Behavior (\t -> do (b1,c) <- bh `at` t
                     s <- getTextMutDoc mutd bl
                     return (s,c))
           (ev -=> ())
```

If we were using the hybrid FRP implementation this would not be possible. Instead we would have to implement a `snapText` function that sampled the document on a given event. By using `addListener` here, we have implicitly dropped the history from the event, and therefore we need not have access to the any of the previous values of the mutable document.

```
snapText :: DocumentB -> Event (TIndex,TIndex)
         -> GUI (Event String)
snapText (DocumentB mutd _) ev = do
  (l,e) <- mkWire
  addListener ev (mapIOL getTextMutDoc l)
  return e
```

The implementation of a purely functional document behavior is therefore still an issue for future research. However, when using the data-driven FRP implementation it costs nothing to instead use a mutable document representation.

8.2. Conclusions

The FranTk widget library that has been implemented in a toolkit independent manner. This should make it relatively easy to port it to other GUI toolkits. Future work will investigate how easy this is in practice. In general, the implementation of FranTk was a significant undertaking, consisting of roughly 12500 lines of code (~2000 for the Fran core, ~7500 lines of code for FranTk, and ~3000 for TclHaskell (the low-level Tcl-Tk binding)).

Part IV. Formal Verification

Part IV of this thesis argues that formal verification can be successfully applied to aspects of an interactive system. It presents a method for deriving a formal specification from a FranTk prototype and demonstrates how this technique was used to prove formal properties about the QOC editor and the ATC system.

Chapter 9 - Previous Approaches to Formal Development

9.1. Introduction

A variety of notations have been developed and used to aid interactive system design. These include graphical formalisms such as Petri Nets [149], tabular notations such as User Action Notation (UAN) [86], and textual formalisms such as process algebras [151] and mathematical specification languages (eg Z) ([1], [85]). Hybrid notations have also been developed, that combine graphical with textual representations [41]. We therefore need to carefully consider criteria for any comparison. This chapter first discusses how specifications can be used either to aid a developer's understanding, or to support verification. We will then consider a selection of different notations and discuss the facilities that they provide. This thesis concentrates on notations which support some form of formal verification. This chapter therefore presents three formal approaches to interactive systems modelling: the LOTOS interactor model, the York interactor model and the ICO Petri net based modelling approach. It will also discuss SpecTRM [120], a notation for modelling safety critical systems, which provides support for both reasoning and verification. It will then discuss the role of formal modelling in this thesis and present a limited set of requirements that fit the goals in this thesis.

9.1.1. Formal Modelling for Understanding

Formal specifications can be used to aid designers in understanding the behaviour of their system. Even when proof is not necessary this can be useful. For instance, User Action Notation specifications have been used simply to understand the behaviour of multi-user systems. For instance, in [171] we demonstrated how UAN could be used to reason about locking strategies in the multi-user QOC editor case study. A variety of locking strategies are possible, some more stringent than others. The semantics of each approach, and the resulting effect on user actions can be considered using formal notations to help determine the merits of each.

Formal specifications can also be related to usability inspection approaches. For instance, the cognitive walkthrough technique helps designers to uncover potential usability problems [208]. Designers can consider a description of the design, along with a description of the tasks that a user needs to perform. This description can be used to look for problems in a design and to see how easily it affords certain actions. A clear and easily understood specification could be useful here as part of the design description. While prototyping and user centred evaluations provide the best way to find usability problems, re-designing a system to remove them requires careful consideration. Lightweight formal modelling can help designers to consider these issues.

Specifications can be used to help designers to consider the predictability of a system. For instance, a specification can be used to consider how each task is performed. We could check that similar tasks were performed in a consistent, predictable way. This sort of analysis relies on a notation being easy for a designer to read. If a requirements specification can be easily understood by domain experts, then it can be used to reason about potential problems in a design; for instance, to consider mode confusion errors within a system [117].

The search for a single specification language to use when constructing a system seems impossible[56]. Instead a range of different modelling languages can be used to consider a design. Maintaining consistency between these different models can then become an important problem. Some form of literate development approach ([29], [101], [19]) providing automated links between different specifications can aid here. One highly-influential commercial modelling language, the *Unified Modeling Language* (UML) has attempted to support this form of approach [163]. Requirements specifications can be constructed in terms of a range of notations including use cases, time diagrams, and state charts. Explicit links can then be defined between each model, making it clear when changes in one will affect changes in another. However, even such integrated approaches are limited. The UML approach does not provide any particular support for specifying user interfaces. Based on a set of studies Jones [105] argues that designers will always step outside the bounds of a given modelling language, when engaged in creative discussion. The development of languages to allow designers to

understand and manually reason about interactive system development therefore remains a significant, and complex research area. It is an area that this thesis will not attempt to address.

9.1.2. Formal Modelling For Verification

A Department of Trade and Industry study looking at formal methods [10] highlighted their inability to handle human-computer interaction as one important reason why they had not received significant uptake in industry. The use of formal modelling to consider interaction issues is therefore important.

Formal specifications can be used to help developers to prove the functional correctness of their systems. For instance, we could prove that a design meets specific, formally defined, critical requirements. This is important because incremental development, based simply on prototyping and testing, cannot guarantee certain critical system properties. When designing a system, there may be millions of possible system states. No amount of testing can significantly test such large state spaces. The problem becomes particularly significant when we wish to prove negative properties about a system. For instance, when designing a rail track control system, we may wish to prove that "A route will never be set if conflicting routes are set" [82]. This sort of critical requirement is usually impossible to prove simply by user testing, because generating a complete set of test cases may be impossible. Though many of these critical requirements are what can be termed functional (unrelated to the interface), many others will be related to user interactions. For instance, when developing an Air Traffic Control system there will be certain interaction requirements that will be critical, such as "A control order can be sent to only one plane" [149]. It is possible to automatically detect certain types of consistency and completeness problems with a specification [120]. For instance, a system can be checked for mathematical completeness, and non-deterministic behavior under the same conditions.

There has also been considerable research into how to prove usability principles, using formal specifications [41]. Various interaction concepts have been suggested. These include *predictability*, whether a system behaves as expected; *visibility*, whether the necessary information is displayed to allow users to act successfully; *continual feedback*, whether a systems provides the necessary feedback to allow users to understand their actions; and *reachability*, whether a user can get to all states in a system, or whether they could get stuck in an interaction deadlock. These principles are important. For instance, an unpredictable system will confuse users. This could lead to dangerous problems in a safety critical environment. However, they are also very general. While they have been formalised, they can still result in meaningless results if applied without great care. For instance, deciding on the necessary information for visibility is obviously a task dependent and often complex process. This thesis therefore concentrates on application dependent verification, rather than on these high level principles.

9.2. York Interactor model

The "interactor" model, developed under the Esprit Basic Research Action *Amodeus* project [41], treats systems as groups of interacting components, each with a state and a behaviour. This is therefore a component based approach, that allows for modular specifications. Designs attempt to relate tasks to interactors and so provide a user centred approach to system modelling. There are two variations on the interactor model, the York and LOTOS variants.

The York model uses a state based approach. The York model (see Figure 43) considers an interactor to consist of an internal state, which is reflected through some rendering relation (`rho`), onto some perceivable representation (`P`). Interactors communicate with the outside world via a set of events. There are two types of event: stimuli, produced by other agents in the environment, and responses, generated by the interactor [85].

.

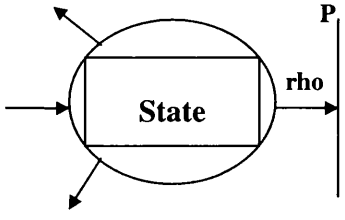

Figure 43 - York Interactor

For instance, consider a simple button. It can be enabled or disabled. This value is changed by the toggle action. When enabled, the button can be selected, by being pressed. It also has a label which can be updated.

The button interactor therefore has a set of events, through which it can communicate with other agents. It has an internal state that is modelled by a group of attributes. The attributes can be altered by events. The behaviour axioms explain how this happens. A statement *P=> [A] Q*, means when *P* is true, if the event *A* occurs, then *Q* will be true afterwards. For instance, axiom 1 says, if the button is *enabled* and the value of *selected* is initially equal to *X*, then after a button press, the value of *selected* will be *not X*. Interactors can then be composed to form more complex interactors.

interactor [button]
Events press,toggle
State attributes label : String selected : Bool enabled : Bool
Behaviour 1 enabled ^ selected = X => [press]selected = not X 2 enabled = X => [toggle]enabled = not X
Rendering output = label ^ (not enabled ^ grayed_out) ^ (selected ^ pressed)

Figure 44 - A York Button Interactor

The development process using York interactors starts by considering the components that make up a system at the highest level. These will, generally, be described using natural language. These components will then be decomposed into lower level interactors, and the internal definitions will be refined to include more detailed and formal descriptions. For instance, in the initial design of an aircraft warning system, we might define two interactors, the pilot and the environment. We could then go on to break down the environment into the components that monitor sub-systems such as engines and hydraulics [55].

The York model makes it easy to consider the state of the system, and the internal description of interactors. It is, however, more difficult to consider how they combine together. Its graphical representation is also relatively poor.

One of the major problems with York interactors was the lack of tool support to allow automated verification. Recent work has begun to overcome this problem. Campos and Harrison([23], [24]) have developed a compiler to convert York interactors into SMV (a model checking tool). The SMV tool specifies a system as a set of state variables and transitions. They have now applied the approach to a few case studies; including a small e-mail client [22], an audio-visual communication system [23] and a simple model of a moded aircraft panel [24]. The last was based on an existing case study by Leveson and Palmer [118]. It did not therefore result in any new discoveries. Though the SMV compiler is a useful tool, it must still be used with care by a formal methods expert. The finite state machine produced by an arbitrary specification can easily become too large to be practical. The specification must therefore be carefully reduced, by removing state variables, and decreasing the size of variable domains. This must be carried out carefully in order not to affect the meaning of the specification.

9.3. The LOTOS Interactor Model

The other interactor model, developed during the Amodeus project, is the LOTOS interactor model (LIM) [151]. It has been developed at CNUCE-CNR in Italy, and uses a process algebra based approach. It views an interactor as an object that can:

- "receive (and accumulate) output from the application side (*oc*),
- receive an output trigger (*ot*), the interactor then sends output to the user side (*os*),
- receive (and accumulate) input from the user side (*im*), and provide feedback toward the user (*os*)
- receive an input trigger (*it*) that causes the interactor to send the accumulated input to the application side (*is*)."

Internally an interactor has four components (see Figure 45). The collection maintains a representation of the model of the interactor. When triggered by the event *ot*, it passes output (using the event *uc*) to the presentation, which performs the actual rendering. Input from the user side is received by the measure which, when triggered, passes it (using the event *md*) to the abstraction. The abstraction alters the input into the form that the application requires; for instance, turning a button click on a menu, into a message saying which menu element had been chosen. The measure may provide feedback on user input, using the event me, for example, to make a button look pressed. The measure may also tell the presentation to modify any new output data. For instance, if a button interactor were told to change its label (by the event *oc*), the measure would be informed (by the event *uc*) and could tell the presentation (using the event *me*) whether the button should be drawn in the pressed state.

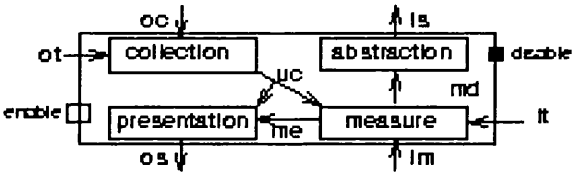

Figure 45 - LOTOS Interactor

We could define a button interactor as follows:

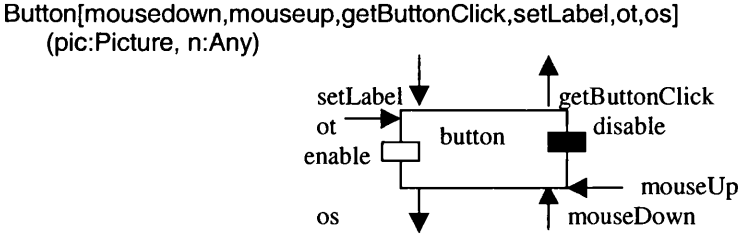

Figure 46 - LOTOS Button Interactor

This receives input which, is either a mouse press (*mousedown*), or a successful click (*mouseup*). Each will cause feedback: the button will appear to be pressed and then unpressed. A successful mouse click is the input trigger, which causes the value n to be sent to the application side. The interactor can also receive output (*setLabel*) that gives a picture to display on the button. The button can also be enabled, so that it accepts input, or disabled, so that it does not.

The LOTOS interactor model provides one further graphical representation to help understand the composition of processes, the *Process Interactor Network* [50]. Here processes are represented by named boxes, communication gates by circles and communication by lines. This further representation complements the graphical interactor diagrams above.

The LOTOS interactor model therefore provides a model that can be used to consider architectural system design.

9.3.1. TLIM – Tasks, LOTOS, Interactors and modelling

This LOTOS interactor model has been integrated into a more complete method for developing interactive systems. This method, the "Tasks, LOTOS, Interactors and Modelling" (TLIM) method, combines the lower level architectural view of the system provided by LIM with a higher level task view [153]. This combined modelling approach can be used at various stages in the design.

It consists of several phases:

1. An initial, informal phase, where user requirements are determined and a task analysis is produced.
2. The tasks are then structured, and described in a hierarchical graphical notation. This notation "ConcurTaskTrees" combines ideas from hierarchical task analysis, with temporal operators from LOTOS. A high level task-based design is therefore possible.
3. This task specification is then transformed into an interactor based architectural description, based on the LOTOS interactor model described above. This model is system based, and considers both the interaction objects visible to the user, and the internal application based objects. This transformation is intended to be semi-automatic.
4. This architectural description can be transformed into a LOTOS specification. The user behaviour can also be specified in more detail in LOTOS. This formal specification can be used to check properties of the design, using automatic model checking tools. Requirements can be defined in ACTL, a branching time temporal logic. These consist of statements about when it is possible to perform certain actions. Designers can then verify whether their specification satisfies these requirements. This sort of analysis can prove particularly useful when designing safety critical systems.
5. The architectural description can also be used to develop a prototype. This implementation could take several forms, and relies on the existence of some user interface programming language. The current CNUCE research is considering the use of object oriented approaches for this phase.

9.3.2. A Brief Analysis

As an international standard, LOTOS has been the focus of a good deal of work on formal verification tools. This provision of automatic tools makes it possible to prove complex interaction safety properties that may be significant to the design. More will be said on these tools in Chapter 11.

The TLIM method provides a behaviour oriented view of system design. It is concerned mainly with the external behaviour of interaction objects and their communication. It uses a "constructive" style of specification, where the effect of events is described. In contrast, the York approach is a "model oriented" approach where relationships are specified, and interactors are defined to obey specific predicates [41]. The graphical notation used in the LOTOS Interactor Model is perhaps clearer than that used in the York interactor model, as it shows how user events relate to application based events, and how they are structured together.

However, LOTOS specifications can be hard to understand at first, and can prove overly complex when dealing with small systems [148]. The current approach to TLIM also lacks real time operators. This problem can, however, be overcome by using newer versions of LOTOS, such as the new E-LOTOS standard [97], which possess real time operators.

More fundamentally, the LOTOS interactor model suffers because it is a purely event based approach. It is therefore more difficult to consider the state of the system. Certain concepts, such as relations between component's states are impossible to define. For instance, a folder window displays a function of the contents of the folder. If we use only an event passing approach, then the folder window component must communicate explicitly with the folder component to maintain consistency between their data. While concepts such as mouse clicks are certainly events and should be treated as such; a mouse position is a status phenomenon. Thinking about such concepts purely in terms of events, for instance a mouse motion event, can make it difficult for a designer to understand a problem properly [38].

The LOTOS interactor model has been applied to a range of case studies including MATIS, an airline booking system, and CERD, a component of an air traffic control system [152]. The LOTOS interactor model is currently undergoing further work as part of the MEFISTO European project, investigating the use of formal modelling approaches in Air Traffic Control .

9.4. Petri Nets and MICO

Palanque and Bastide [12] have developed an integrated approach based around Petri nets and object oriented design. This method, the MICO method, uses Petri nets to produce both a task and a system model. These models can be automatically combined, to check for conformance between them. Designers can therefore check that the system model supports all necessary tasks. This approach relies on the Interactive Cooperative Objects (ICO) formalism. This formalism borrows concepts from the object-oriented paradigm, including dynamic instantiation, inheritance and client/server relationships.

The ICO model was originally designed to handle event driven interfaces. The behaviour of a system is described using Petri nets, while the components are described as objects. The actions that can be performed in the Petri net, are therefore supplied by object definitions. The ICO model uses timed Petri nets to allow real-time concepts to be discussed. Petri nets describe a system in terms of state variables (named places, shown as ellipses), and by operations (called transitions, shown as rectangles). These components are connected by arcs. The system state is given by the marking of the net, shown by a distribution of tokens on the net's places. State changes result from the firing of transitions, resulting in a new token distribution. To fire a net, all input places must be marked. Then tokens are removed from the input places and are deposited on the new output places.

The ICO formalism is also being developed as part of the MEFISTO project. Recent work has extended the ICO formalism to cover other aspects such as services, state and presentation. An ICO offers a set of services that define the programming interface offered by the object to its environment. The state of an ICO is the distribution and the value of the tokens in the object's Petri Net. Because services are related to transitions, the current state may influence the availability of services, and conversely the performance of a service will alter the state. Rendering functions can be associated with places, thereby specifying what effects a dialog change has on an interface. Recent work has begun to link Visual Basic prototypes to ICO models. This allows changes to the dialog petri-net to be viewed as the interface is used.

The MICO approach has a number of positive features. Again system requirements can be specified in ACTL and proved using automated tools. A study by Palanque, Paterno et al [148], suggested that Petri Nets provide a good way of representing small-scale descriptions of systems, and of representing abstract views of a system's behaviour. However, they perform more poorly than LOTOS when describing more detailed specifications. The composition operators in LOTOS make it easier to build large, detailed, modular systems [148]. The MICO method also suffers from some of the other disadvantages associated with the LOTOS model, as it is also an event based approach. MICO objects are used to describe components, we therefore have some notion of internal state. However, it is still not possible to consider status phenomena.

9.5. SpecTRM Requirements Modelling

Leveson has been working on the development of a formally based approach to requirements specification [120]. SpecTRM (Specification Tools and Requirements Methodology) is a CAD system for digital automation. It is intended to aid engineers in developing complex, safety critical systems. It emphasises early discovery of errors, through early analysis; recording requirements and design rationale; and reuse in system design. It provides a set of tools to support a range of different forms of analysis, including safety analysis, test data generation and completeness and consistency analysis. It uses a state-machine requirements modelling language based around a state charts variant. A black box modelling approach is used. That is, models describe component behavior only in terms of outputs and inputs that stimulate or trigger those outputs. A model does not include any information about internal design, only about externally visible behavior. This allows the construction and analysis of systems a compositional manner; review and analysis of the specified behavior of a component can be separated from review and analysis of the internal design and implementation of the component.

The language has been designed explicitly with readability in mind. The graphical and tabular nature of the language makes it easy to show to engineers. The language is, however, based on an underlying formal model, the Requirements State Machine. Using this formal model it is possible to check automatically for a number of basic properties, such as completeness, and non-determinism. They define requirements completeness as "the specification being sufficient to distinguish the behavior of the desired software from that of any other, undesired program" [88]. To guarantee this, each state must have a single (and therefore deterministic) response for every possible input. A specification can be analysed in small pieces, allowing incremental construction and analysis. Models are, therefore, defined in terms of hierarchical state machines. To prevent a state space explosion, all of the analysis is performed directly on the model without generating the full finite state machine for the system. They define a set of restricted composition operators for RSM. Individual components can be analysed for basic properties. A set of rules have been defined which determine if these properties are maintained when hierarchies, parallelism and event propagation are introduced.

The approach has been applied to a number of major case studies. RSML (SpecTRM's predecessor) was applied to TCAS II, a complex, airborne, collision-avoidance system required on all commercial aircraft with more than 30 passengers that fly in US airspace [88]. The automated tools were able to show a non-determinism in the TCAS specification that was unplanned, unobvious and had serious safety implications. SpecTRM was also applied as part of a NASA funded safety analysis of Air Traffic Control upgrades [119]. The SpecTRM modelling and analysis approach has therefore proved to be useful and scalable.

However, the range of properties that can be tested for automatically with SpecTRM is still very limited. Other verification approaches, such as model checking, are more powerful, allowing more complex properties to be checked. However, scalability is much more of an issue with these more powerful verification approaches.

9.6. Requirements for Formal Modelling

There are, therefore, a range of formal modelling approaches that can be applied to interactive systems. Formal models can be used either to help developers reason about and therefore understand a system and to allow automated verification of important properties. Both of these are complex areas in their own right. This thesis focuses only on support for automated verification. As a result, issues such as the readability of a notation can be ignored. There have been two classes of approaches discussed in this chapter: (1) restricted, but easily scalable, verification approaches, such as those provided by SpecTRM; and (2) powerful, but more complex verification approaches as supported by interactor modelling approaches. The latter are explicitly based in formal, mathematical languages. Their use is, therefore, geared towards formal methods experts. This thesis adopts the latter approach, and considers its use for checking domain and task specific problems, rather than for guaranteeing higher level usability properties. We are therefore interested in approaches which *support formal verification, of complex, domain specific properties by formal methods experts.* Any approach that wishes to support this goal must therefore satisfy a number of requirements.

9.6.1. Verification Tool Support

The language must have good tool support. This support must allow specifications to be checked for both validity (i.e. syntax and type checking), but must also support some form of formal verification, such as model checking.

9.6.2. Link to Prototype

It must be easy to develop a specification based on an interactive system prototype. This should preferably be supported by some form of transformation tool, or compiler. This compiler should provide good support for partial verification of a system, because any realistically large system will be far too large to model check in one go.

9.6.3. Applicability

Using the approach it should be possible to find problems with the specification. By using only partial verification it is very difficult to argue that we have *proved* that a system is safe. In contrast, *partial verification is only really useful if it discovers problems that would otherwise not have been found.*

This thesis therefore considers formal methods to be simply another mechanism for finding problems with a system.

9.6.4. Scalability

Any approach should be applicable to large systems. It is only in the development of such systems that realistic and useful problems will be found.

This thesis builds on the LOTOS interactor work. It presents a method that supports the creation of a formal, LOTOS, specification, which given certain parameters can be derived automatically from a structured FranTk prototype. The model can be analysed to verify important safety properties about the system design. Chapter 9 will discuss the transformation process; Chapter 10 will discuss the application of LOTOS model checking technology to the case studies.

Chapter 10 - Deriving a Formal Specification

10.1. Introduction

As we saw in the previous chapter, a system can be formally modelled in LOTOS as a network of interacting components or "interactors". An interactor is represented as a process that has a state, and accepts updates, that change its state. It also receives requests that make queries about its state. As discussed in the previous chapter, this interactor modelling approach has been applied to a range of application areas, including Air Traffic Control. Developing LOTOS interactor models requires a considerable amount of effort. The ability to automatically generate them is, therefore, important.

We could have attempted to analyse the original Haskell program using Equational Reasoning. This approach allows us to prove equality between different functions, by simplifying them using equational laws. However, it cannot be easily applied to applications that make extensive use of input/output. Thomson [199] applied basic modal logic to Fran, allowing him to reason about temporal properties. However, in his approach all reasoning was done manually. In contrast, a LOTOS model can be analysed, using model-checking tools, such as the CADP toolset [66].

The state space explosion problem makes it totally impractical to attempt to generate the entire state space of any realistically large system. We therefore instead provide support for generating models of manageable components of a system. Verification must (and should) therefore be focused only on critical aspects of a system.

An interactor network is similar to the architecture model discussed in Chapter 5: an Abstract Behavior is similar to a LOTOS interactor. This chapter will first introduce the basics of LOTOS, and will then show how to convert from a FranTk architecture to a LOTOS interactor model.

10.2. Overview of LOTOS

LOTOS is a standardised Formal Description Technique for the specification of concurrent systems. It consists of two sub-languages, a data part and a control part.

10.2.1. ACT ONE

The data part is based on algebraic abstract data types. Data is specified in the ACT ONE specification language. A data type is described in terms of a set of operations, and a set of equations. As an example consider a partial definition of the Boolean type.

```
type Boolean is
  sorts Bool
  opns false    : (*! constructor *) -> Bool
       true     : (*! constructor *) -> Bool
       not      : Bool -> Bool
       _and_, _or_    : Bool, Bool  -> Bool
  eqns forall x, y : Bool
    ofsort Bool
      not (true)  = false;
      not (false) = true;
      x and true  = x;
      x and false = false;
      x or true   = true;
      x or false  = x;
endtype
```

The Boolean data type defines the sort *Bool*. It has two constructors *true* and *false*, and a set of operations. (In reality, the Boolean type would need more than just the four operations given). Some of these are infix operators shown as *_a_*. The meaning of these operations is defined in terms of a set of

equations that describe their relationships. These equations are used to rewrite a compound expression until it becomes a constructor. Data types can import other types. The ACT ONE language is monomorphic, strict and first-order, (i.e. functions cannot be passed as values.) These restrictions have to be overcome when transforming a polymorphic, lazy, higher order FranTk program into LOTOS.

ACT ONE is a relatively expressive language. It is, however, fairly cumbersome to use. Its syntax can be clumsy, for instance, it does not support if-then-else expressions. It has no primitive data types. Instead, all data must be defined in ACT ONE. This means that there is no syntactic sugar for types such as strings. As we must enumerate every possible constructor, it makes it impossible to define continuous types such as *Reals*. Some LOTOS tools, such as those discussed in the next chapter, allow ACT ONE data to be linked in with C code. A mapping must, however, still be produced between each C value and ACT ONE constructor.

The differences between FranTk and ACT ONE data make defining a mapping between them a complex problem. More will be said on this translation process in Section 10.3.3.

10.2.2. The Control Language

The control part of a LOTOS specification is based on process algebra, combining features of CCS and CSP. A concurrent system is considered to be a collection of processes that communicate synchronously; that is they rendezvous at particular points. LOTOS has an interleaving semantics. This means that concurrent sets of actions are interleaved together to form a sequence. Two actions cannot therefore be carried out at exactly the same instant in time. LOTOS specifications are composed together using the operators in Table 1.

stop	An inactive behaviour, representing deadlock
G !V ?X:S ; B	Interact on gate G, sending value V and receiving a value of sort S henceforth to be referred to as X. Then behave as process B.
B1 [] B2	Behave as B1 or B2, whichever starts first
[E] -> B	If E is true then behave as B
B1 |[G1,..Gn]| B2	B1 in parallel with B2, synchronised on gates G1 .. Gn (||| means no synchronisation, || means full synchronisation)
hide G1 .. Gn in B	make gates G1 .. Gn invisible from the outside, these actions are replaced by the internal action i
exit	successful termination
B1 >> B2	B1 followed by B2, when B1 terminated successfully
B1 [> B2	behave as B1 until either B1 terminates or B2 performs its first action, after which point behave as B2
P [G1..Gn] (V1..Vm)	Call process P, with gates G1 .. Gn and value parameters (V1 .. Vm)

Table 2: LOTOS Operators

10.2.3. E-LOTOS

The new E-LOTOS standard replaces ACT ONE with a functional language for describing data. This would reduce the semantic jump from Haskell. E-LOTOS, however, still uses a first order language so some of the same changes would still be required. The E-LOTOS standard also simplifies the control syntax, and adds explicit real-time support. This addition, in particular, would have been useful when transforming real-time FranTk programs into LOTOS.

One of the reasons for choosing LOTOS was the assumption that E-LOTOS tools would have become available quickly enough to use during this thesis. Unfortunately, though verification tools have now begun to appear [182], they appeared to late for use in this thesis. This thesis therefore only discusses translation into LOTOS.

10.3. Converting FranTk into LOTOS

10.3.1. Generating a specification

An interactor network is similar to the architecture model discussed in Chapter 5. We can consider a LOTOS interactor to be equivalent to an Abstract Behavior. Both have a state, and accept updates from the user side. They also both receive values from the application side via requests.

We can translate a FranTk architecture into an interactor network in three stages.

- Turn each Abstract BVar into an interactor
- For each subview relationship a parent component can send disable and enable events to its children.
- Link requests and updates between components.

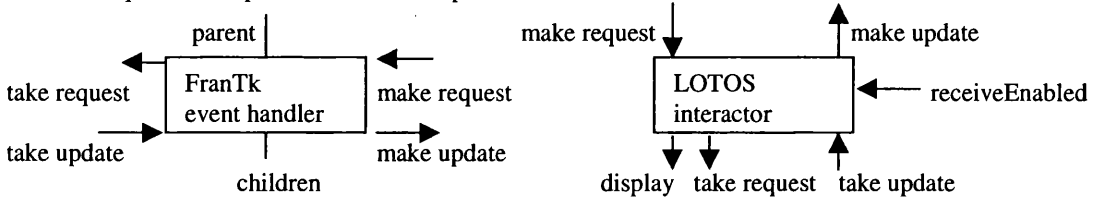

Figure 47 - Relationship Between FranTk and Lotos Interactors

There are two significant differences between the LOTOS specification and the FranTk architecture. Firstly, FranTk implicitly updates values through behavior requests. Programmers must explicitly send updates up the tree (towards the application). FranTk then sends request values back down the tree (towards the user), after evaluating constraints. In LOTOS this must be modelled as a synchronisation phase, where updated request values are propagated around the interactor network. For a distributed, multi-user system, each distributed component has a separate synchronisation phase.

Secondly, if we later wish to verify properties about our specification we need to produce a finite LOTOS specification, that is a specification with a finite number of states. LOTOS model checking tools, such as Eucalyptus discussed in the next chapter, produce a state transition graph equivalent to the LOTOS specification. To be able to do this we need to be able to enumerate every possible transition, and therefore every possible gate and value that will be produced. As requests and updates become LOTOS gates, the set of values that they actually use must therefore be finite. An interactor specification must also have only a finite number of instances of any given component.

Translating arbitrary Fran component definitions into LOTOS can be a very complex procedure. However, we can easily handle the subset of behaviours that have been defined as state transition functions. Concentrating on this subset of the prototype may seem restrictive. However, the high level behavioural models, such as the aircraft trajectory in the ATC system, can be understood relatively easily and are not amenable to model checking. State transition functions involve far more conditions and, at least in the case studies undertaken in this thesis, turned out to be found at the critically complex areas of the design. For instance, with our ATC system, the critical areas of interest are the data link communications, both between sectors and between aircraft and controllers.

10.3.2. Transforming Abstract BVars into Interactors

The transformation of a Clock abstract data type into a LOTOS interactor is done in two parts. We must provide an interactor process to describe the behaviour and an ACT ONE data type to describe the state of the object. The relationship between an Abstract BVar and a LOTOS interactor can be seen in Figure 48:

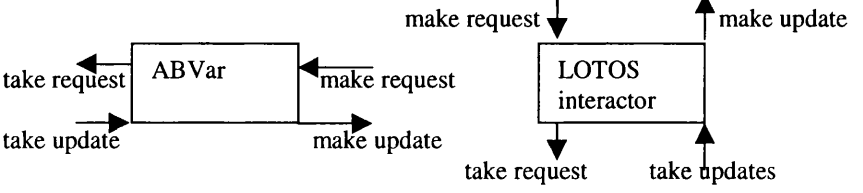

Figure 48 - Relationship Between an ABVar and LOTOS Interactor

As an example, consider the clearance interactor in the ATC system. It has two methods, a request that accesses the current clearance value, and a listener update that is told of DL message updates. (This information is available from the Architecture model).

```
ABVar: ClearanceFL
requests:clearanceState :: ClearanceFL -> Behavior ClearanceFL
update: clearanceMsg :: Listener (DLMsg (Cond,Int))
```

The `ClearanceFL` interactor is modelled as shown below.

```
process ClearanceFL [clearanceMsg,sample] (f:ClearanceFL) : noexit :=
        clearanceMsg?m:DLMsgPairCondInt;
        ClearanceFL [...] (handleClearance (m,f))
        []
        sample!f; ClearanceFL [...] (f)
endproc

process RunClearanceFL [clearanceMsg,sample]:noexit :=
      ClearanceFL [clearanceMsg,sample] (initClearance)
endproc
```

The definition can be read as follows. The `ClearanceFL` process handles two events (*clearanceMsg* and *sample*), and has a state (*f* the current clearance value). It can receive a *clearanceMsg* event, accepting *(?m)* a message, after which it continues, updating the value of the flight level clearance *(f)*. Alternatively *([]* choice operator), it can synchronise with a sample request sending out *(!f)* the current clearance state. In general, we generate one gate for each listener update. In addition we generate a single sample gate, which returns the entire state of the interactor. Note that behavioral values can be replaced with static ones, because the entire interactor state will be resampled every time it changes. Finally, we have a run process which calls the interactor with the initial data value.

10.3.3. Translating Haskell into ACT ONE

If an ABVar is defined as a state transition function, we can produce ACT ONE data type descriptions for the equivalent Haskell data types and functions. ACT ONE supports overloading, so that operations can have the same name, but full polymorphism is not supported. We must therefore generate a data type for each instance of a polymorphic Haskell data type. For instance, part of the automatically generated ACT ONE definition for the *Clearance* data type is shown below.

```
type ClearanceFL is PairCondInt, DLMsgPairCondInt, ...
sorts ClearanceFL
opns
   Cleared (*! constructor *) : PairCondInt -> ClearanceFL
   Clearing (*! constructor *) :
      PairCondInt, MsgId, ClearanceStatus, PairCondInt -> ClearanceFL
   handleClearance : DLMsgCondInt, ClearanceFL -> ClearanceFL
eqns ...
handleClearance (Clearing (val,,msg,WaitingLack,oldval),
                 Response Lack ref) =
       Clearing val msg WaitingResponse oldval
endtype
```

A few further changes are in general required:

- Curried Haskell functions (such as `f a b`) become uncurried ACT ONE operations (i.e. `f (a,b)`)
- We must convert lambda expressions into separate functions.
- We must convert if-then-else and case expressions into equations.
- We must turn higher order functions into first order ACT ONE expressions.

The first and second of these are trivial. The third can also be done fairly simply.
An expression of the form, `if-p-then-a-else-b` becomes the equation p => a; not (p) => b.

An expression of the form `case x of {p1 -> a;p2 -> b;...}` becomes a new set of equations f (p1) = a;f (p2) = b; ...

The last of these is more complex. Haskell supports functions such as *map*, that take others as arguments. It also allows functions to be partially applied. In ACT ONE neither of these is possible. Instead we generate a data type with constructors for each use of partial evaluation, and an application function that will be called when arguments have been passed to a function. For instance, consider the following example Haskell code. It demonstrates the use of higher order functions. The *apply* function performs simple function application.

```
module Test where

apply :: (a -> b) -> a -> b
apply fun val = fun val

f :: Int -> Int
f x = x + 1

result :: Int
result = apply f 1
```

For ACT ONE, we generate a type, `Int_Int`, to handle the higher order function. We have one function argument, *f*. We therefore require one constructor to cover the use of this function. Note that the type of the constructor is `Int_Int`, matching an `Int->Int` function. When this function is finally applied, we replace its use with `applySrt`. This takes a constructor, and value and determines what to apply to the value based on the constructor. The `apply` function therefore simply needs to call `applySrt`. The `res1` function then simply calls `apply` with the constructor for `f`.

```
type Test is Int
sorts Int_Int
opns
    fCon (*! constructor *) :  -> (Int_Int)
    applySrt : (Int_Int),Int -> Int

    apply : (Int_Int),Int -> Int
    f : Int -> Int
    res :  -> Int
eqns forall v2_val:Int,v3_x:Int,v1_fun:(Int_Int)
    ofSort Int
    applySrt (fCon,var1) = f (var1);

    ofSort Int
    apply (v1_fun,v2_val) = applySrt (v1_fun,v2_val);
    f (v3_x) = plus (v3_x,One);
    res = apply (fCon,One);

endtype
```

This shows the basic approach to handling higher-order functions. Whenever we have a function value, we replace it with a constructor and an application function. We can sometimes avoid the need for this sort of generation by inlining definitions involving higher order functions. For instance, if we inline the `apply` function, (replacing all uses by its definition) we remove the need for a higher order function in this example.

When generating ACT ONE code, it can be very easy to generate intractably large LOTOS definitions. We noted earlier that to allow automated model checking a LOTOS specification must contain a finite number of states. When transforming an ABVar into LOTOS we must be able to enumerate every possible request that the ABVar may offer, such as every possible clearance value in the `ClearanceFL` interactor. Clearly if the Clearance definition relied on large or even unbounded types such as integers or strings, we would not be able to generate a finite number of states. We must use a similar approach to that used by Campos and Harrison in their Interactor->SMV compiler[23]. We must reduce the domain of state variables. We do this by replacing types with enumerated data types. For

instance, we could replace the Integer type with an enumerated type specifying a small set of integers actually used in the ABVar. There are often parts of a data type that are irrelevant for a given test, in these case we can enumerate only a single value. These assumptions *must* be recorded and justified by the developer when applying this approach. When converting a FranTk program to LOTOS, the developer provides a set of definitions which override those in the actual FranTk program. These definitions can include functions as well as data type enumerations.

When overriding, type synonyms may be considered to be separate types. A type synonym inherits the enumeration of its original type unless a more specific one is provided. For instance, if we had the type synonym `type Label = String` in a FranTk program, we might well wish to provide separate enumerations for Labels and Strings.

```
data String = One | Two

data Label = Lbl1 | Lbl2
```

If FranTk programs make good use of type synonyms this can significantly reduce the size of the state space. If the example above did not use a type synonym then all of the necessary values would have been required in the enumeration of String.

We must therefore be careful when transforming Haskell code into ACT ONE. There are a number of important issues:

- syntactic differences such as currying, conditional and lambda expressions
- polymorphism and parameterised data types v simple overloading
- higher order functions v first order functions
- the generation of finite types.

10.3.4. Verifying Parts of a System

Instead of generating a LOTOS specification for the entire ATC system, which would be intractably large, we can select individual components and produce a smaller specification of only the critical elements of the system. Again the developer must document and justify the choice of these elements. This selective generation can be done in one of two ways. We can select an individual abstract behaviour and produce a LOTOS model of it alone. Alternatively we can transform a subtree of the architecture into an interactor network.

10.3.4.1. Verification with a Single Interactor

By analysing individual Abstract Behaviors, we can carry out analysis early on, when only a small part of the whole system may have been implemented. A control process is generated, which runs in parallel with the abstract behaviour process, and repeatedly performs one of any valid update actions followed by a sample event. These valid events are defined as a constraining function of all available input events. For instance, for a given test we may wish to constrain the test to use only a restricted set of the available updates. Alternatively, we may wish to constrain the possible update values. For instance, with the `ClearanceFL` interactor we would want to restrict ourselves to a finite sequence of clearance events that have consecutive message identifiers.

10.3.4.2. Transforming an Interactor Network

We can generate a LOTOS specification for a tree of components. When doing this we can again define a constraining process, to run in parallel with each ABVar process, which restricts the available updates to an ABVar. The use of such constraining processes is necessary to allow compositional model generation, discussed in the next chapter (section 11.3.5).

Here we must restrict the behaviour of the interactor process so that it maintains the communication order and concurrency control used in FranTk. In FranTk we can consider evaluation of user input to happen in three distinct phases. Firstly, after some user input, all resulting updates are sent via listeners and events to the appropriate BVars. This will be referred to as the *update phase*. Secondly, the system

then samples all Behaviors. This results in propagation down the architecture tree. It will be referred to as the *request phase*. Thirdly, the view of each component is recomputed and redisplayed in a *redisplay phase*. On start up, the system performs one request and redisplay phase, thereby sampling the initial state of each component. FranTk also supports a system clock which provides the current time, and ticks just before every sample.

We run the interactor network in parallel with a control process. This starts and stops each phase and generates a tick update before each sample. Any time based interactor, such as one that depended on a time based predicate would listen to this tick update. The system clock is modelled as an interactor, which is updated by this tick, and which provides access to the current time as a request gate.

This means that we need to restrict the behaviour of the LOTOS interactor network so that updates, requests and redisplays happen in these three distinct phases. To satisfy this we make use of input and output triggers to restrict when interactors may receive input. These triggers are used by each interactor to synchronise with other interactors in the interactor network. The relationship between a FranTk event handler and a LOTOS interactors can be seen graphically in Figure 49.

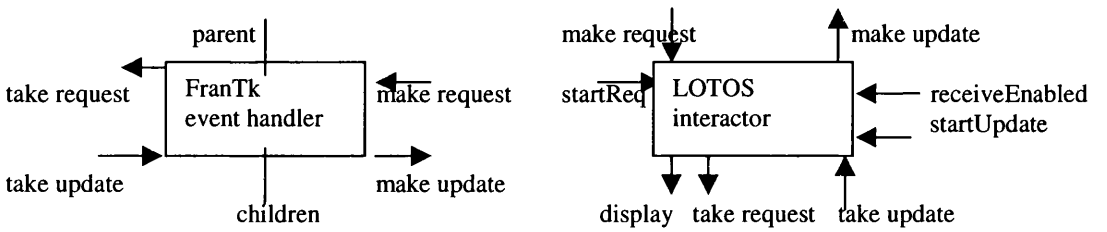

Figure 49 - Restricting LOTOS interactor behaviour

Rather than provide a translation of the view function of each component, transformation is considered to take place on abstract interaction objects. That is, a tree of Abstract BVars is considered to have a user interface which consists of a set of update methods (representing user input) and request methods (representing display updates). The developer selects a set of request methods which they are interested in, and a set of update methods that will be considered as user actions. They can also specify a set of other methods in which they are interested; all others will be considered to be internal events. LOTOS provides the ability to hide internal events, thereby allowing us to make observable only those events in which we are actually interested. When generating the finite state machine, we can reduce it with respect to observational equivalence. This produces a considerably smaller state space, in which only the observable events remain (but with the original ordering guaranteed) [111].

When displaying an interactor tree, a component must enable and disable instances of its children where appropriate. We could dynamically create a new process every time we create a new child and then destroy that process when the child is deleted. However, the ability to recursively and dynamically create processes allows for the existence of an infinite number of processes. LOTOS verification tools will often have difficulty with this form of behaviour. Many, such as those in the Eucalyptus tool set used in the next chapter, forbid this style of specification. Instead we must create a finite, given number of child processes at the start and then disable and enable these when necessary. This requires us to enumerate the complete set of children that may ever exist, and produce an instance of the child process for each. We must also provide the algorithm with the complete set of instances of each component that may exist. This list of instances of each child is provided as input to the activator. Finally, a component may only enable its children if it is itself enabled.

10.4. Implementing the algorithm

This approach has been used in a small prototype compiler, implemented in Haskell. The compiler requires several input parameters. Firstly, it must be told which subset of the architecture tree to transform. Secondly, it expects an input file containing any necessary overriding definitions for functions and data types in the system (this includes, for instance, enumerations of unbounded datatypes). Thirdly, it expects a restriction file that restricts the updates available to each interactor. Finally it expects a file enumerating all the components in the system (as each component must have a finite number of children).

It then carries out the transformation in several phases:

- It parses the set of relevant Haskell files;
- It finds each relevant ABVar, and its set of update and request methods.
- It generates code for the set of methods, and for each data type and function used by this set of methods.

The compiler currently only supports a basic subset of Haskell 98. It supports components written in a simple state-transition style, and it does not include support for advanced features such as type classes. *The compiler was developed as a proof-of-concept prototype; it was not intended for, and has not been used by any other developers.*

10.5. Discussion

This chapter has described an approach that can be used to transform parts of a FranTk program into a LOTOS interactor specification. The algorithm has been implemented as a prototype compiler in Haskell. The approach provides a rapid way to generate formal specifications from a prototype.

However, the transformation must be applied very carefully. It can only be applied to elements of a FranTk program written in a state-transition programming style. Care must be taken to generate finite, small specifications. It is very easy to generate intractably large specifications using the approach. The compiler can therefore only be sensibly used by a formal methods expert, with a sound knowledge of LOTOS, and a reasonable understanding of the intricacies of the transformation approach. It is not, therefore, a panacea that allows formal methods to be used by non-specialists; it is simply a tool for use by formal methods specialists in the development of specifications for a prototype interactive system.

Chapter 11 - Performing Formal Verification

Formal analysis can be used to verify completeness criteria about user interaction, to search for paths to hazardous states that might be reached within an interface, and to verify consistency questions about interaction when in different modes of a system. Mode confusion can be a serious problem in complex systems [176]. Studies have shown that with aircraft automation pilots can lose track of automation behaviour and perform the wrong action, an error of commission. In more complex settings, errors of omission can be dominant. Here the operator fails to take a required action, often because a system has done something undesirable, perhaps because of an unnoticed mode change.

Leveson et al [117], Javaux [96] and Rushby [168] have shown that state based modelling can help in finding "accidental complexity" within system designs that make mode confusion errors more likely. They suggest a number of different criteria for analysis. Many of these criteria require designers to manually analyse a specification. A LOTOS specification is not sufficiently readable to make this practical. However, some criteria can be analysed automatically. In particular, we can search for inconsistent behaviour in an interface, where similar tasks or goals are associated with different actions.

Given a specification it is important to be able to reason about it. This can, for instance, be done, by simulating the specification to check that particular sequences of actions are possible (e.g. [198]). We cannot, in general, prove with such a technique, that a particular sequence of actions will not happen. We may also not be able to prove that a sequence of actions is possible with only finite resources. However, this is often very important.

Model checking provides a more powerful approach for carrying out such verification. For instance, the use of model checking tools to test for problems such as mode confusion is currently a popular area of research (e.g. [24]). Given some specification that can be translated into a Finite State Automaton, such as a LOTOS specification, and some temporal logic formula, a model checker can perform a fully automated proof of whether the specification satisfies the formula. This makes it easier to perform complex proofs in the context of iterative design. Changes to a specification will require only that the temporal formula be checked automatically against the new specification. This form of rechecking can be as automatic as using regression tests in software design. This is in contrast to Theorem Proving, where a whole series of lemmas may be required, before a formula can be completely proven again.

However, model-checking environments also tend to have some disadvantages. In general, they can only perform proofs across finite systems, using finite types. We therefore need to produce a LOTOS specification with a finite state space. For instance, with our QOC editor specification we proved properties about an instance where there are only a finite, and given number of users, nodes and edges in existence. The generation algorithm discussed in the previous chapter produces such a specification. Even if we have produced a finite specification, the state space explosion problem may still make model checking impractical. The problems arise when trying to convert a specification into a finite state automaton. It can be well beyond the available computing resources to enumerate the entire state space of a specification. Solutions to these problems do, however, exist. We can for instance use compositional model generation [53] to generate the state machine in several steps.

There has been some other work using model checking with interactive systems specifications before. Paterno [152] used LITE, a LOTOS model checker, to verify properties about interactor specifications written in ACTL, an action based temporal logic. However, LITE only worked with basic LOTOS; that is on LOTOS specifications that did not make use of ACT ONE data. Specifications could simply be produced using the core LOTOS operators outlined in Table 2 (on Page 172). This restriction reduced the generality of the approach. In contrast, this chapter applies model checking technology for full LOTOS to verification of interactive systems.

In this chapter, I will firstly give a brief introduction to simulation techniques, and explain why they are not capable of performing all necessary forms of verification. I will then go on to consider model checking with the Eucalyptus toolset [66] for LOTOS. I will discuss model generation approaches that can avoid some of the state space explosion problems. I will argue that compositional model generation, in particular, is well suited to interactor specifications, and show how it can be automatically supported

by the FranTk-LOTOS compiler discussed in the last chapter. I will then show how model checking, with the μ-calculus (a modal logic), can be used to prove important properties about a specification.

11.1. LOTOS Simulation

LOTOS simulation tools, such as LOLA [43], can be used to perform some reasoning and verification with a specification. For instance, [198] has shown how to use LOLA for a form of verification known as property testing. We can attempt to check safety properties, which state that something bad should not happen. Given a specification, we can define a dangerous trace of actions as a LOTOS process, which ends with a *testUnsafe* event. We can compose this unsafe process in parallel with the specification, to form a test process. If the *testUnsafe* event is reached by this test process then we have determined that the specification is unsafe.

For instance, we could define the locking property for the QOC case study as follows.

```
process unsafe [lock,lock,testUnsafe] :exit :=
  lock?user:String?id:String;lock?user2:String [user ne user2]!id;testUnsafe;exit
endproc
```

It says that if one user can lock an object, and then a second user (where user does not equal user2) can lock an object, then we have reached an unsafe state.

We can use LOLA to check whether we can reach the unsafe event. If a specification contains non-terminating processes, as all specifications produced by the algorithm in the previous chapter will, then an expansion depth must be specified. This restricts how much of the state space must be explored. If the undesired behaviour occurs within the given depth we have proved that the specification is unsafe. However, if the undesired behaviour does not occur, then we have simply shown that our specification may be safe with a degree of confidence determined by the expansion depth.

Thomas [198] shows how to expand a combined test process, to try to prove that the test is never passed. If we have a process that has a finite number of states, but that never terminates we can transform the test specification into a set of recursive equations and then examine the non-recursive prefixes to ensure that the test event does not occur. This approach, however, requires us to find these recursive prefixes by hand. This means that we must also reduce our specification to a manageable size by hand, extracting only the parts of our specification that we wish to reason about.

Markopoulos [127] used this sort of approach to demonstrate task conformance between a task definition and a system specification. He generated a number of action sequences that represented important activities in the task model, and used LOLA to try to demonstrate that these action sequences were possible in the system specification. This approach can therefore demonstrate that the temporal ordering, defined in the task model, is maintained by the system model.

The advantage of using simulation tools such as LOLA is that we can specify processes that can have an infinite number of states, and therefore that may use non-finite data (such as all the natural numbers), or that may dynamically create new instances of a particular process. This is because we only expand a process to a particular depth. However, we can only reason about processes up to that particular depth. Even given a process with a finite number of state, we may still not be able to verify that a particular trace does not occur. More advanced verification tools and techniques are therefore necessary.

11.2. Model Checking

The Eucalyptus toolset [66] provides an interface to a set of LOTOS tools. An important part of it is CADP (Caesar Aldebaran Development Package) which provides a model checker for LOTOS. It provides facilities for transforming a LOTOS specification into a transition graph known as a labelled transition system (hereafter referred to as an LTS), and for proving properties about this LTS using a variety of temporal logics.

It consists of several parts:

- CAESAR which compiles a LOTOS program into a transition system. The ACT ONE data part is first compiled into C code by CAESAR.ADT which is used to compute this graph.

- ALDEBARAN is a verification tool for comparing or reducing specifications according to bisimulation relations.

- OPEN/CAESAR - consists of several tools for exploring transition graphs. It includes a tool (XSIMULATOR) for simulating a specification, and tools for on-the-fly examination of a LOTOS specification.

- XTL [128] is a programmable temporal logic checker. It uses a special functional programming language which can be used for graph exploration. XTL supports definitions for temporal logics, including the μ-calculus used in this chapter.

EUCALYPTUS provides an X-Windows based, graphical user interface to allow access to this set of tools (see Figure 50). The left side of the interface provides an icon for each file. For each icon a pop-up menu provides access to the set of possible operations. The right side of the interface contains a text window showing the output from the operation. This provides a fairly basic, but simple interface to the toolset. As model checking tools are fully automated, and so require no user involvement during verification, this form of interface is sufficient. More sophisticated interfaces are needed for interactive theorem proving where user involvement in the proof process is significant [3].

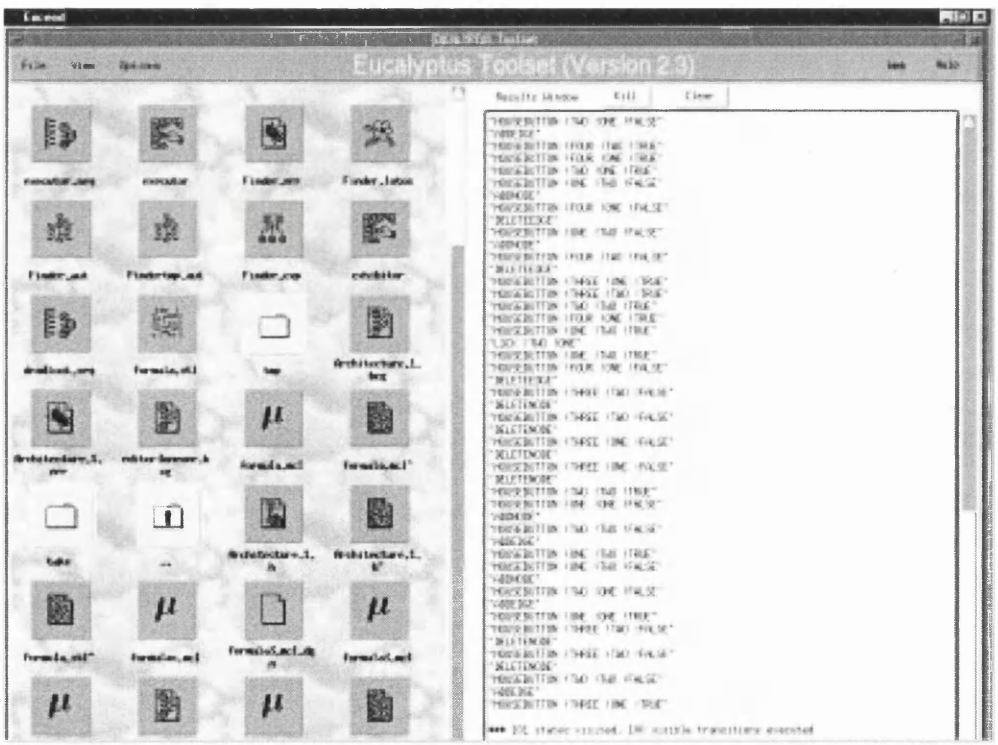

Figure 50 - The Eucalyptus Interface

The EUCALYPTUS tools have been used on a variety of significant case studies. These include specifying distributed systems; network protocols; hardware protocols such as the PowerScale, a multiprocessor architecture based on PowerPC processors and used in Bull's Escala servers and workstations [26]; and embedded software, such as the flight warning system of the Airbus 330/340 [65].

I will now discuss model generation, and then model checking with the CADP toolkit.

11.3. Model Generation

The semantics of a LOTOS specification is described in terms of a labelled transition system (LTS). The CADP tools can generate the LTS for a finite LOTOS behaviour. This means that dynamic creation of processes is impossible because this would lead to possibly infinite behaviour. It is, therefore, not permitted to write expressions such as:

```
process X = P ||| X
```

CADP is also incapable of dealing with expressions that make unrestricted use of types with an infinite number of values, such as integers. This is because, when producing a transition graph for a specification, CADP will have to enumerate every possible transition. For instance, if we used an integer type then performing the gate `X!1` would be possible, but attempting to perform `X?Integer` would require CADP to generate an `X` transition for every integer. The generation algorithm discussed in the previous chapter produces a finite specification.

However, even with a finite specification it may still be impossible to produce the LTS, if the specification is too large. A specification can easily expand into a graph with many millions of possible states. The maximum number of states that can be handled by CADP is dependent on the amount of memory in the given workstation. Using around 30Mb of RAM Aldebaran can handle only a few million states. We therefore need more sophisticated approaches to model generation. These minimise graphs with respect to bisimulation relations, and include such as compositional, symbolic and on-the-fly graph reduction techniques.

11.3.1. Model Minimisation

There are a number of relations that can be used to compare LOTOS expressions. We can make use of these relationships to reduce a large transition graph into a smaller one. For each behavioural relation, R, the minimisation of a given LTS S with respect to R, involves finding the smallest LTS (in number of states) which is R-equivalent to S [111].

Though a variety of relations exist we will consider two here, observational equivalence (a coarse relation) and strong bisimulation (the finest relation). Both these relations maintain safety properties: so that if a process S deadlocks, then its minimised form will also deadlock. This holds because the minimised form of S may only perform a sequence of events, if S can also perform that sequence of events.

For two LTS, L1 and L2 , to be equivalent, then if L1 performs a sequence of actions L2 must be able to perform the same action, and vice versa. The restriction on which sequence of actions to consider marks the difference between observational and strong equivalence. With observational equivalence, we care only about sequences of observable events. LOTOS considers all hidden events to be unobservable; unobservable events are represented by the internal event *i*. Occurrences of this internal event *i*, are therefore ignored. With strong bisimulation if L1 performs the internal event *i*, then L2 must also perform *i*. We can define this formally as follows:

Definition 1: a labelled transition system L, is a quadruplet L=(Q,A,T,q0) where Q is a finite set of states, A is a finite set of actions, T is a transition relation ($T \subseteq Q \times A \times Q$) and q0 an element of Q called the initial state. We denote a transition $(p,a,q) \in T$ by $p \xrightarrow{a} q$.

Definition 2: For each relation $R \in Q \times Q$, we define:

$$B_L(R) = \{(p1,p2) \mid \forall l \in L,\ (\forall q1 . (p1 \xrightarrow{l} q1 \Rightarrow \exists q2 . (p2 \rightarrow q2 \wedge (q1,q2) \in R)) \wedge (\forall q2 . (p2 \xrightarrow{l} q2 \Rightarrow \exists q1 . (p1 \rightarrow q1 \wedge (q1,q2) \in R))\}$$

The bisimulation equivalence $\approx_L$ for the language L is the greatest fixed point of B_L.

Depending on how we define L we can produce different relations. Strong bisimulation is defined when $L = \{\{a\} \mid a \in A\}$; observational equivalence is obtained when $L = i^* \cup \{i^* a i^* \mid a \in A\}$.

Reduction with respect to observational equivalence is particularly useful when considering interactive systems. Though a system may have a complex internal behaviour, we only need to consider those input and output events observable to the user. In the previous chapter, the transformation algorithm produces a specification where a small set of events (those which we wish to reason about) are visible, and all others are hidden. When producing the LTS of such a specification, if we reduce the model with respect to observational equivalence, then we can produce a much smaller LTS. It will contains only those events about which we wish to reason, but will have the same behaviour as the full specification with respect to those events. For instance, with the QOC editor, the LTS after reduction had around a thousand states; in contrast the full transition system would have consisted of many millions of states.

11.3.2. Symbolic Minimal Model Generation

As discussed earlier, it is often not possible to produce the entire graph of a LOTOS expression because it becomes unmanageably large. However, when reduced with respect to a bisimulation relation we may get a LTS that is of a manageable size. One approach to solving this problem is therefore to produce the graph and minimise it simultaneously. This is known as minimal model generation [53]. This form of generation can be very effective when the size of the reduced state space is small but the full state space contains many millions of states.

11.3.3. On-The-Fly Techniques

Another effective way of handling large specifications, is to use on-the-fly analysis [54]. Caesar can produce an abstract representation of a LOTOS specification, rather than enumerating the whole state space. Given such an abstract representation, we can attempt to perform verification with the abstract model. In this case, the LTS is generated as it is needed to perform the verification. This can be very effective if we only need to check a part of the whole LTS to perform this verification. CADP allows verification from simple deadlock detection, to comparisons of LTSs and evaluation of μ-calculus formulas all to be done on the fly.

11.3.4. Compositional Model Generation

CADP also support compositional generation of a LTS. For a LOTOS specification with a number of parallel processes, we can generate the LTS for each process. These can be reduced with respect to a bisimulation relation, before being composed together to form the whole system. This approach can be more effective than either of the two previous techniques [111], handling larger specifications, and doing so far more rapidly.

This sort of approach is particularly applicable to interactive system specifications. These consist of sets of abstract components, which, are themselves, formed from sub-components. For instance, in the QOC editor, a window work area consists of a set of nodes and edges. These abstract components contain internal state, and therefore behaviour, which will be hidden at higher levels. In particular, an interactor specification generated by the algorithm in the previous chapter will be formed from a hierarchical tree of processes. We can therefore generate the complete LTS for the specification by generating the LTS for each component at a given level of the tree and then composing these together at the level above.

Compositional model generation must, however, be done carefully. In a constraint-oriented specification, such as the ones produced by the FranTk-LOTOS generation algorithm, two processes `P || Q` will strongly constrain each other's behaviour. Generating `P` or `Q` separately may produce a far larger graph than their composition [111]. This will certainly be the case in a generated interactor network, where, for instance, an ABVar interactor will actually receive only a small number of updates from the restricted set of components in existence, but which will in theory be able to receive a far larger set of updates.

This problem can be overcome by synchronising `P` or `Q` with an environment `E` ([154], [111]). This environment E should be defined such that for every event, `e`, in `P`, if `P || Q` can perform `e`, `then P || E` must be able to perform `e`. The process `E` is therefore a "conservative approximation" [154] of the rest of the system, as seen from `P`. That is, `E` allows all executions that `P` can go through as part of the whole system. This solution can be applied very easily to an interactor specification generated from FranTk. An automated approach for supporting this will be presented in the next section.

Where necessary the three approaches can be combined. For instance, we could generate all the separate processes in a specification, and then compose them using minimal model generation, or verify the composition of transition graphs on-the-fly. When developing the Thesis case studies the two most useful techniques appeared to be compositional model generation and model minimisation.

11.3.5. Automated Support for Compositional Model Generation

The FranTk-LOTOS compiler provides some automated support for compositional model generation. This is very important as Pecheur [154] argues that dividing a specification into suitable components, and restricting these components with a suitable environment process, can be a very complex task.

This automation is done in two stages. We can restrict every interactor so that it only accepts the updates, that would be provided by the system. We can then restrict every interaction object process so that it only accepts requests that will be provided by these restricted ABVars. A list of possible updates must be given to the FranTk-LOTOS compiler for every ABVar to be restricted. This is done using the restriction file mentioned in the previous chapter. We can then generate the LTS for the restricted ABVar. From this, we can automatically extract all possible *sample* actions and form a *restrictsample* process. Finally, we can generate a process call for every component instance, and restrict it with respect to all relevant requests.

The complete generation of an LTS will therefore be done in 4 phases:
1. generate the LTS for each ABVar interactor;
2. generate all the restriction processes for each component interactor from these LTSs;
3. generate the LTS for each interaction object component;
4. finally, generate the complete LTS, by composing the LTS hierarchically up the tree.

At each stage we reduce a given LTS with respect to observational equivalence, before composing it with its parent.

This approach is very effective. It maps well onto component based interactive systems specifications. This has allowed it to be fully automated using the FranTk-LOTOS compiler. It will allow generation of far larger specifications than are possible without compositional generation.

11.4. Model Checking With The μ-Calculus

Once we have generated the LTS for a given specification we can perform a variety of types of verification. For instance, we can automatically search for any deadlocks with Aldebaran [67]. We could also compare a specification with a test process as discussed in section 11.1. However, a more "powerful" way of verifying properties is to evaluate temporal logic expressions against the LTS. In this section we consider one such logic, the μ-calculus, a branching-time modal logic. Most propositional temporal and modal logics used in computing are sub-logics of the modal μ-calculus [214]. In it we can express a range of properties covering liveness (such as "eventually an action will happen"), safety (such as "it is always possible to do an action") and fairness (such as "infinitely often an action can happen").

11.4.1. The Basics Of The μ-Calculus

The μ-calculus is a branching time modal logic that can be used for model checking [66]. It is used to express the capacity to perform some action. Consider the following LOTOS process:

```
P1 = A;B;P1 [] C;P1
```

It perform the events `a` then `b`, or `c` forever. We can consider two properties that are true for this definition. These are:

`<"A">T` - In the current state (i.e. at first) it is possible to perform action `A`
`["B"]F` - In the current state (i.e. at first) it is not possible to perform action `B`

Informally, <x>P means that there exists an *a*-transition from the current state, and after it P is satisfied; [x]P means that after all *a*-transitions from the current state, P is satisfied. Here a transition is an event such as A, B and C in the process P1.

In general, we express the capacity to perform some action, *a*, as <a>T and incapacity as [a]F. We can also express that any action is possible or impossible <.>T and [.]F. The former therefore means that we will not deadlock, and the latter means that we will.

We can summarise the behaviour of <x> and [x] in the following table. The first column represents the case when the desired action is available, the second when the desired action is not available.

	Action x is available	Action x is not available
<x>T	True	False
<x>F	False	False
[x]T	True	True
<x>F	False	True

Table 3 - The Behaviour of <x> and [x]

11.4.2. Fixed Points

So far we have expressed formulas that hold only over the current state. We need, however, to be able to express formulas that hold over some future state, possibly even over every state. This can be done by making use of two important operators, *gfp* (greatest fixed point) and *lfp* (least fixed point). These operators take the form gfp X . p X and lfp X . p X where X ranges over a family of propositional variables. (The gfp operator can also be written as ν, while lfp can be written as μ.)

They are best understood by example. If we wish to state that it is never possible to perform an event D we can do this as follows:

```
gfp X . (["C"]F and [.]X)
```

This is a recursive definition. It says that in the current state it is not possible to do a D-event and, after any transition [.] the definition still holds (i.e. it is not possible to perform a C).

If we wish to express that in some future state it will be possible to perform a B-event, we can use the following:

```
lfp X . (<"B">T or <.> x)
```

This says that in the current state we can perform a B, or after some action (<.>) the definition is True (i.e. it may now be possible to perform a B).

It is significant that one of these definitions uses the gfp operator, and the other uses lfp. They have subtle, but significantly different meanings and often give different results when used. Intuitively lfp should be used to express *eventualities*, that is something may eventually become true, gfp should be used to express necessity, that is that some condition must always be true. The gfp will produce the largest process that satisfies the given expression, while lfp will produce the smallest. The gfp can sometimes return the infinite process as a result while lfp may return the empty process. The table below shows when to use both definitions.

	lfp	gfp
f X = <a>T or [b]X	use	Infinite process may be generated
f X = [a]F and [b]X	null process may be generated	Use

Table 4- Fixed Point Operators

11.4.3. Simplifying Specification

Using fixed point operators, we can express a wide range of properties. However, fixed point operators can be difficult to read and understand when they become nested. To overcome this problem, a number of macros can be used. (These form part of the LTAC temporal logic that is defined in terms of the μ-calculus.)

For instance, we can express that a process never deadlocks, or that it is always possible to perform a particular action using the ALL operator:

```
ALL (<.>T) says that it is always possible to perform some action, that is we never deadlock
ALL ([″D″]F) says that it is never possible to perform an event D
ALL (<″A″>F) says that it is always possible to perform an event A

(ALL (p) = gfp X . p and [.]X.)
```

With respect to process P1, the first two of these statements are True, and the last is False (as after we have performed an A, we can only perform a B). The ALL operator is therefore used to express safety properties, that may be important when verifying an interactive system, such as "At any point a control order can only be sent to one plane".

There are several other macros that are useful. We can state that we will inevitably reach a state where we can perform an A using the INEV operator:

```
INEV (<″A″>T) says that inevitably we will be able to perform an A
INEV (<″B″>T) says that inevitably we will be able to perform a B

(INEV (p) = lfp X . P or [.] X)
```

When applied to process P1, the first of these formulas is True, as in the initial state we can perform an A. The second of these formulae is False, however, as we may continue to perform the action D forever, and so because we never perform an A, we will never be able to perform a B.

The inevitably operator can therefore be used to express liveness properties that state that we can eventually do some action. This can be important when verifying interactive systems, as we need to express requirements such as "when a command is sent it will eventually reach a plane, and after this there will be some feedback".

We can state that in some future state it will be possible to perform an A, using the potentially (POT) operator:

```
POT (<″A″>T) says that in some future state we will be able to perform an A
POT (<″B″>T) says in some future state we will be able to perform a B

(POT (p) = lfp X . P or <.>X)
```

When applied to P1 both these formulae are True. We can therefore use the potentially operator to express reachability concerns with an interactive system, saying that after some point it will be possible to perform an action.

Finally, we can express that some expression is True until some event using the until operators. Two such operators exist, strong until (SU) and weak until (WU). Strong until is equivalent to weak until, except that the future change event must occur. For instance:

```
WU ["B"]F <"A">T says that we cannot perform a B until such time as an A occurs, but an A
need not occur
SU ["B"]F <"A">T says that we cannot perform a B until such time as an A occurs, additionally
at some point we must perform an A
```

```
(WU (p1,p2) = gfp X . p1 or [.]X)
(SU (p1,p2) = lfp X . p1 or [.]X)
```

The *until* operators are useful as they allow the expression of interaction properties such as "A message may not be sent to the second plane, after sending it to the first".

11.4.4. Verification with XTL

Verification of μ-calculus formulas is done using the LTS-specific functional language XTL [128]. This language allows the definition of macros. Several temporal logics have been defined in it including the μ-calculus. XTL provides functions for handling states, transitions and labels, and sets thereof. It can match gates, and be used to do pattern matching on the labels of transitions. For instance, the following function searches for some transition of G with integer parameter larger than 10 [154]:

```
exists T : edge where
    T -> [G ?X : integer where X > 10]
end_exists
```

This ability to carry out pattern matching is very powerful, and allows us to write more general formulas than is often possible with model checking. For instance, the older μ-calculus tool, Evaluator, is only capable of matching gates with explicit-value data. The current transformation algorithm used in CADP to produce a LTS loses some of this type information. Instead of providing access to the complete set of ACT ONE sorts and equations defined in the specification, all values are translated into one of three types, booleans, integers or, if neither of the above is appropriate, strings. Future versions of CADP should, however, overcome these restrictions.

11.5. Formal Analysis in the Case Studies

We applied formal analysis to both the QOC editor and the ATC case study. With the first of these we were able to prove some simple properties. For instance, we verified that our system implementation supported the locking properties that were highlighted in the design:

- No two users can alter the same node at the same time;
- If one user has locked a node it cannot be deleted by another user, even with delete group.

However, these checks were fairly trivial; we did not find any problems with the system. The ATC case study provided more interesting results.

The ODIAC working group, at EUROCONTROL, has specified a number of data-link communication protocols. When developing our prototype, some of these protocols were available as transition diagrams, however, some were only available as natural language descriptions. We therefore wanted to verify that our prototype correctly, and robustly, supported these protocols, and that our interpretation of the natural language descriptions was consistent and complete.

We analysed the prototype with respect to each of the three major communication protocols: transfer between sectors, negotiation of transfer parameters, and flight clearances. We used a mixture of simple simulation, to step through the specification, and model checking to carry out the analysis. We compared the behaviour of the system under these communication activities to check whether errors were handled consistently. For instance, if several messages were to be sent in quick succession and some of them failed, would error messages creation, and future system behaviour in response to controller commands, be consistent.

We found one significant problem in the prototype. We do not claim that using formal verification was the only way to find this problem; other approaches, such as state-chart based walkthroughs may also have found it. However, it had lain undiscovered in the code despite a reasonable amount of testing.

The following modal logic formula specifies that always (`ALL`) after an ATC Communication Management message is sent, the message will inevitably be displayed as accepted (`DISPACPT ! m`) or a failure message will be displayed (`DISPFAIL ! m`).

```
ALL ([] (SENDACM ?m:Msg,
         INEV ((<> (DISACPT! m))
               or
               (<> (DISPFAIL !m))
              )
        )
    )
```

This formula did not hold true, and CADP provided a counter example. The problem occurred if two messages were to be sent by the controller in quick succession. If a logical acknowledgement message were to be received for the second message before the first message timed out, then the time out on the first message would be ignored. This would result in the system continuing to wait for a reply and so neither an accept nor a fail message would be displayed to the controller.

This verification complements formal approaches such as Leveson's Requirements State Machine (RSM), used in SpecTRM [120]. The LOTOS specification is at a similar level of abstraction to Leveson's SpecTRM-RL. Both provide black box models of a system, concerning themselves with the externally visible behaviour of system components and their interacting behaviour.

Leveson's approach is more general. It covers all aspects of the requirements process. It also allows a global analysis of the entire system, for properties such as completeness criteria. The modelling language used in SpecTRM is also far more readable, as it uses a mixture of graphical and tabular notations. However, our LOTOS specification provides a powerful model of the user interface of the system. As it is derived from the prototype, it reduces the time necessary to develop a formal specification, and is well integrated into the development of the interactive components of a complex system. Though it is not very readable, it can be executed, allowing specialists to step through the specification. It can be used to automatically verify more complex properties of a system than is possible with SpecTRM.

11.6. Conclusions

It can be very important to perform verification with a specification. Model checking provides an effective way of doing this. The Eucalyptus toolkit provides a powerful set of tools that can be used to carry out model checking with LOTOS. We have to be careful when generating a state model from a LOTOS specification. It can be difficult to avoid the sate space explosion problem. However, compositional model generation provides one powerful way of doing this. We can support this form of model generation in our FranTk-LOTOS compiler by producing extra code, and restricting the behaviour of each component automatically.

The μ-calculus, supported by XTL, provides a powerful tool for model checking. It can be used to express a wide range of temporal properties that we may wish to hold true about a LOTOS specification. The verification with the ATC system did prove useful finding an error in our prototype. However, the verification tools must be used with great care. Their use requires a working knowledge of temporal logic; logic formula must be applied with care, as not all will return a result in a sensibly small period of time.

Part V. Conclusions

Part V of this thesis presents conclusions about both the FranTk GUI library and about the formal verification work. It also highlights areas requiring future work.

Chapter 12 - Conclusions and Further Work

12.1. Background Summary

The focus of this thesis was on the development of tools to support prototyping and verifying interactive systems. The creation of complex, multi-user systems is a difficult process. It requires an iterative development approach. Such systems must allow group awareness and support co-ordination and communication between different users. These users must be able to understand the common context that they are working within. This means that usability heuristics will be even more difficult to use in a CSCW project. Only through thorough evaluation can we hope to develop systems that truly support their users. A good example of such problems arise in the development of Air Traffic Control systems. Social studies of such systems [84] have shown that co-ordination between Air Traffic Control Officers is subtle, complex, and often outwith the bounds of regulated procedures; the usability of these systems can be in direct contrast to standard usability heuristics.

12.2. FranTk Contributions

There is therefore a need for good prototyping tools, that support the rapid development of complex, interactive systems. The first and most significant contribution of this thesis, is therefore the presentation of FranTk, a new declarative prototyping language for creating complex, dynamic interactive systems. It has been applied to a range of case studies including two multi-user systems.

FranTk draws on ideas from both Clock and Fran. It concentrates on providing a programming model that is both "declarative in the large and in the small". This allows systems to be built in a high level, structured manner. In particular, it provides good support for specifying real-time properties of such systems. This thesis has discussed the development of a fully fledged user interface library. It has been released as a publicly available toolkit (http://www.haskell.org/FranTk).

FranTk was developed to satisfy a set of high-level requirements. It should be:

- High level and declarative;
- Support declarative concurrency;
- Provide a compositional programming style;
- Support good application/interface separation;
- Provide tool support where appropriate;
- Be scalable, and therefore applicable to large systems;
- Be efficient enough to handle large systems.

FranTk provides a number of major contributions that satisfy these requirements:

- FranTk lifts Fran's behaviors and events to widgets. This is the key to the declarative style of programming. The appearance of a widget can be defined *for all time* in terms of FranTk combinators. An interface can therefore be defined as a function of some application state. At no point do we require to imperatively define *how* an interface will change; instead, we simply state *what* it should look like.

- FranTk provides good support for dynamic as well as static interfaces. The construction of systems with a dynamically changing number of components can be difficult in many GUI systems, and frequently requires a very imperative and sometimes cumbersome style of programming. FranTk provides dynamic collections. These can be used to model dynamic applications, such as sets of aircraft, in a declarative style. They can be rendered efficiently in an incremental manner. The use of behavioral values and dynamic collections allows a single abstract model of an application to be produced. We can then have multiple views of this model, providing good application/interface separation. This separation is particularly important in the development of multi-user systems.

- FranTk extends Fran with support for hierarchical interactive displays, allowing access to input from individual components rather from one monolithic window. This was vital to allow a truly compositional style of programming.

- FranTk separates visual composition from semantic wiring. These two concepts are fundamental to GUI programming. The first involves geometric composition. For instance, placing one widget above another. The second involves connecting user input from a widget to the application code.
  - This separation is made possible by the introduction of *listeners*, consumers that respond to user input. FranTk provides an algebra to compose these listeners in a functional style.

  - This separation allows individual user interface components to be represented by untyped values, we can therefore compose collections of components. While there is a separation between top-level windows, standard components, and canvas components, this is necessary because each of these represent a separate class of widget; it does not make sense to geometrically compose a top-level window and a button beside each other. In addition, because FranTk components are actions which produce widgets, they may have internal state. However, components may be treated as values, they can be geometrically composed used pure functions. We can attach listeners to the user input from a component, including a composite component; we can also apply style configuration options to a composite component. We therefore have a very compositional programming style.

- FranTk clearly supports declarative concurrency. The ATC system, for instance, consists of a number of concurrently evolving components (such as aircraft). These are modelled in terms of behaviors and events rather than requiring any explicit pre-emptive concurrency. FranTk provides support for explicit concurrency where required; however, the support for declarative concurrency was sufficiently powerful to support two different multi-user interfaces.

- FranTk provides a more efficient implementation of the core Fran combinators. This thesis has presented three possible implementations of the core combinators. Each implementation relied on two key features for efficiency.

  - *Data Driven Behaviors and Events.* A simple implementation of events and behaviors requires that behaviors and events are sampled every time interval. This would be prohibitively expensive in a large user interface, as every aspect of the interface would need to be redisplayed every time any input was received. Instead FranTk, uses a data driven model. Events and behaviors have invalidation actions associated with them. When the listener talking to the event or behavior is fired the invalidation action is performed. After any user input only those components that rely on behaviors or events that have been invalidated need to be redrawn.

  - *Weak Listeners and Finalisers.* Once a BVar has been created, the listener will begin talking to the BVar, passing on every value it hears. This is useful only so long as the event or behavior from the BVar is in use. However, often these will only be used for a fraction of the lifetime of a program. For instance, if a component were later removed from the screen and the behavior it relied upon was no longer used it would be useful to remove the listener as well, to prevent unnecessary work. For this purpose, we use weak references. Weak references allow us to add a *finaliser* to an object, which is an action to be run when the object is garbage collected. This mechanism is used to delete listeners. When the clients (the events and behaviors) that a listener can talk to are all garbage collected, a *finaliser* will be run to delete the listener.

- FranTk makes two further implementation contributions.
  - *Incremental Dynamic Collections.* The implementation of efficient dynamic collections requires some extra sophistication. A collection behavior is considered to consist of two parts, a simple behavior representing its value at any given time, and an event generating individual incremental changes. To allow the definitions of functions such as map, and to ensure efficiency, each value is associated with a unique identifier. These unique identifiers and associated incremental updates are generated through a static, incremental collection data type.

- *Toolkit independent implementation.* Though FranTk has been implemented on top of Tcl-Tk, the widget set has been implemented in as toolkit independent a manner as possible. This should therefore allow it to be ported to other toolkits at a later point.

- We have presented two visual tools for use with FranTk. The first is an architecture editor which can be used to help structure large FranTk programs. The second is an interface construction tool which can be used to visually construct and integrate static widgets. These were both developed as proof of concept prototypes to demonstrate that such tools could be easily integrated into a declarative GUI language such as FranTk.

FranTk has been evaluated through a number of significant case studies. In particular, the ATC system provided a very effective test of the scalability and applicability of FranTk. It makes use of most of the features provided by FranTk. In particular, the system makes use of dynamic collections to model aircraft and datalink messages. It includes a number of different views of each of these collections. The ATC system also makes heavy use of real-time predicates, to support concepts such as message time outs.

The ATC system design was based on a set of requirements, discussed with a human factor's (HF) specialist, at the UK's National Air Traffic Services. The system underwent a process of redesign with the HF specialist. This allowed us to investigate whether FranTk was capable of supporting any requested changes. It allowed us to test whether these changes could be carried out rapidly enough to support rapid iterative design. Finally, it allowed us to test whether the quality of the resulting prototype would be sufficient for the needs of real users. We were able to carry out a number of important changes. The HF specialist was impressed by the speed with which changes were made, the interactive nature of the process and the quality of the resulting interface. We were able to develop a tactical data entry widget which has since been recommended as the primary method of tactical message composition in EOLIA (the NATS datalink project). The close user involvement therefore provided an important element of realism to the evaluation. We were able not only to demonstrate a successful application of FranTk, but to produce a prototype widget that has been successfully adopted by the end-user organisation.

12.3. Formal Verification

Formal specifications can be used to help develop interactive systems. Formal analysis can be used to verify completeness criteria about user interaction, to search for paths to hazardous states that might be reached within an interface, and to verify consistency questions about interaction when in different modes of a system. This thesis does not attempt to use formal methods to understand the usability of a system. Instead it concentrates on the use of formal methods to help verify critical, application specific requirements. In particular, it takes one restricted view of formal methods. It assumes that they should be used to find problems in a system design; not to somehow prove that the system works. Verification provides one mechanism of increasing confidence in a system, but it only becomes really worthwhile if it can be used to actually find hidden problems. In Chapter 9 we discussed two classes of approaches: restricted, but easily scalable, verification approaches, such as those provided by SpecTRM; and powerful, but more complex verification approaches as supported by interactor modelling with heavy weight formal methods. The latter are therefore explicitly based in formal, mathematical languages. Their use is therefore geared towards formal methods experts. This thesis adopts the latter approach, and considers its use for checking domain and task specific problems, rather than for guaranteeing higher level usability properties. This thesis concentrates on providing an approach which *supports formal verification, of complex, domain specific properties by formal methods experts.*

In the field of formal modelling of interactive systems, this thesis makes one contribution:

- It presents a method that supports the creation of a formal, LOTOS, specification, which given certain parameters can be derived automatically from a structured FranTk prototype. This allows the generation of a formal model at relatively low cost. The model can be analyzed to verify important safety properties about the system design. To make the verification practical we focus on partial verification, focusing on critical areas of the design. This avoids the state-space explosion problems faced when trying to perform exhaustive proofs about a whole system.

This approach has been evaluated using the ATC case study. The need for significant case studies was very important. Only through the use of a significant, safety critical case study, such as the Air Traffic Control system, can the utility of such an approach be demonstrated. Using the case study we were able to find a previously undiscovered bug in our ATC prototype. The verification tool therefore proved a useful addition. However, we do not claim that this problem could only have been found through heavy-weight formal methods. Other approaches, such as state chart based walkthroughs may also have found the problem.

12.4. Problems and Future Work

There are a number of important issues which remain to be addressed. These issues can be separated into three areas: the design and evaluation of FranTk, the implementation of FranTk, and the formal verification work.

12.4.1. FranTk Design & Evaluation

- There are a number of language extensions and tools that could prove useful for FranTk programmers. In particular it is currently unclear how to use the new Haskell Exception mechanism within a FranTk program. There is also little explicit debugging support within FranTk. More powerful debugging tools are currently an important area of research within the Haskell community. It is not immediately clear how easily the internal machinery of the FranTk implementation could be hidden when using such a tool. One interesting area of future research would be to investigate how easily such tools could be used in conjunction with a high level toolkit such as FranTk.

- This thesis has purposefully said very little about the usability of FranTk. A competent study of the issues was outside its scope. Section 6.5.3 briefly discussed some informal reactions from functional programmers who have used FranTk. Though reactions were generally positive there are clearly some problems in the usability of the language. The use of listeners seems to be an initial conceptual hurdle. Their use was a fundamental design choice. It allows separation between semantic wiring and geometric composition. However, they also introduce an imperative feature into the otherwise declarative model. It remains to be seen whether there is a better way of handling them. Further work is required to properly evaluate the usability of FranTk, in order to determine where problems exist and to either simplify the conceptual model or provide better tool support to overcome them.

12.4.2. FranTk Implementation

- This thesis presented three FRP implementations. Neither of the first two were perfect. The first implementation was a purely data-driven implementation. This implementation is efficient and robust. Unfortunately, it requires a change in the type of any combinator that relies on an event's history. The data-driven representation does not store an event's history and therefore any such combinator must be an IO action. In FranTk, this is not a particularly problematic restriction, as all FranTk programs use the GUI monad. The second implementation was a hybrid that combined the streams and data-driven approaches. Unfortunately, this implementation while generally faithful to the Elliott and Hudak formal semantics of FRP is not entirely robust. In particular, the use of `merge` is not always referentially transparent. This implementation therefore serves more as an example of the difficulties that can arise when implementing FRP. The third implementation presented in this Thesis looks, perhaps, the most promising. It is a refinement of the basic data-driven implementation, which attempts to satisfy the Wan-Hudak semantics. Further research is required to determine whether this approach will actually work. *The development of a truly efficient, robust implementation that is faithful to the formal semantics of FRP therefore remains a topic for further research.*

- A number of other aspects of the implementation are still incomplete. Most significantly FranTk does not include a purely functional document implementation. Instead it uses a mutable (C based) document model. The implementation of a purely functional document behavior is therefore still an issue for future research.

- Though providing acceptable performance, FranTk is still fairly memory intensive[30]. Its performance is usable for a prototyping environment. However, it remains to be seen whether it could be optimised sufficiently for use in real products.

- Though two of the case studies were multi-user systems, they each had a fully centralised implementation using the X-windows client-server model. This is because there is currently no distributed version of Haskell available. In contrast, by using its structured architecture, Clock provides automated support for providing semi-replicated implementations of a system [74]. It would be interesting to see whether similar techniques could be applied to a FranTk program, given that they rely on similar architectures.

12.4.3. Formal Verification Work

- The LOTOS transformation algorithm represents only an initial step towards providing specifications of interactive systems. The approach as outlined in this thesis has a number of serious limitations.
  - It allows only the transformation of sections of FranTk code written in a simple state-transition style. This leaves out large sections of a FranTk program.
  - It must be used with great care, in order to generate tractably small specifications. We have not provided anything other than a set of very high level guidelines for use of this approach. The development of a real methodology to support use of this tool is one possible future area of research.
  - We have not provided a proof that the transformation algorithm really generates a correct model of the system. This remains another possible future addition.
  - The use of LOTOS as the verification language is itself questionable. It was chosen because of the existence of the LOTOS interactor model, and its previous application to interactive systems. However, basic LOTOS has a number of restrictions, such as no support for real-time specifications. It was hoped that robust tools for the new E-LOTOS language, which does provide real-time operators, would have become available. A transformation to E-LOTOS remains an area for future research. It is also likely that there are better target languages available to perform verification on interactive systems. The choice of the best specification language remains an area for future research.

- This thesis has focused on providing links to heavyweight mathematical languages, in particular LOTOS. One interesting area of research would be to provide instead a link to more readable notations such as Leveson's SpecTRM.

[30] The ATC simulation runs in between 30 and 40 Mb of memory, when compiled with the Glasgow Haskell Compiler (GHC). Though significant, this is arguably within the same ball park as Java's Swing.

Part VI. Appendixes

There are three appendixes in this thesis. Appendix A introduces the basic concepts found in Functional Reactive Programming: Listeners, Events and Behaviors. Appendix B contains a brief discussion about usability evaluation in the design of interactive systems. It discusses a small evaluation that was carried out when developing the QOC case study. Appendix C discusses an earlier attempt to link formal specifications with functional GUI languages.

Appendix A Functional Reactive Programming Combinators

This appendix provides a basic introduction to the 2 key concepts in Functional Reactive Programming: *Events* and *Behaviors*. It presents the algebra of operators available for each. The other fundamental concept introduced by FranTk is the *Listener*. This appendix briefly discusses Listeners. However, the algebra of Listener operators is discussed in Section 4.5. Many of these FRP operators are unnecessary for day-to-day FranTk programming. However, they can be extremely important and they form the basis of the implementation of types such as BVars.

Chapter 7 discusses 2 different semantic models for FRP and presents 3 different data-driven implementations. The types of the operators in this appendix fit with the first implementation discussed in Section 7.2. Some of these operators therefore have different types than in the original Fran distributions.

A.1 The Basic Concepts – Behaviors and Events

To begin with, let's consider the basic definitions.

A.1.1 Listeners

A `Listener` is an abstract type, but it can be thought of as `Listener a = a -> GUI ()`. A value of type `Listener a`, is a function, that given a value of type a, performs a side-effecting GUI action with it. Listeners are therefore *consumers* of values. Recall from Section 4.5 that this has important consequences for how the listener algebra is structured. The types seem to be reversed as we apply a function to values that a listener is about to receive before passing them to a listener to be consumed.

A.1.2 Events

`Events` are used to model things that happen at discrete points in time, such as mouse clicks. `Events` can be thought of in terms of a declarative or an imperative semantics. In declarative terms, an `Event` is a stream of *occurrences*, each of which has a specific value and a time that it occurred. The type `Event a` therefore denotes a source that generates values of type `a` at each occurrence.

Events are *producers* of values. Because Events produce values and Listeners consume them, we can connect them together. This results in the imperative interpretation of Events. From this standpoint, an Event is something that we can add Listeners to. These will perform actions on every event occurrence. Listeners and events meet with `addListener`.

```
addListener :: Event a -> Listener a -> GUI Remover
type Remover = GUI ()
```

The function `addListener` adds a listener to an event. The remove action that is returned will then delete that listener when necessary at a later date. Events therefore serve client Listeners.

The name `Event` is perhaps confusing because an `Event` is not a single occurrence, but a stream of or a source for occurrences. A better name might be `EventSource`; however, the name `Event` remains for historical reasons.

A.1.3 Behaviors

The final fundamental concept is the `Behavior`. A `Behavior` is a continuous value that changes over time. A value of type `Behavior a` is a time varying value of type a: i.e. it is conceptually a function from `Time -> a`.

Behaviors therefore *have values*. The value of a behavior can be sampled (snapshotted) on an event occurrence or when a listener is fired. Events can be used to build *reactive behaviors* which change course in response to events.

A.2 What can we really do with Events

This section begins by presenting the basic Event algebra. It then goes on to discuss the more problematic issue of history based combinators. The algebra of operations available on events resemble closely those available for listeners (Section 4.5.2 discussed this Event-Listener duality).

A.2.1 The Event Algebra

We can make an Event that never produces any occurrences using `neverE`.

```
neverE :: Event a
```

We can merge two Events using `mergeE`. This forms a new Event by merging the occurrences of both argument events based on their occurrence times. There is also an infix version of this function `.|.`. Here we see the first of a number of infix operators that make up the Event algebra.

```
mergeE,(.|.) :: Event a -> Event a -> Event a
```

Again there is a list version of merge for events

```
anyE :: [Event a] -> Event a
anyE = foldr (.|.) neverE
```

We can map functions over events using `mapE`. This applies the given function to every occurrence of e to yield a new Event.

```
mapE :: (a -> b) -> Event a -> Event b
```

There are also infix operator versions of `mapE`, including a version that ignores the value produced and just uses an alternative constant value.

```
(==>) :: Event a -> (a -> b) -> Event b

(-=>) :: Event a -> b -> Event b
e -=> b = e ==> \_ -> b
```

We can apply a filter function to events in a similar way to listeners. These functions all filter the occurrences produced by an event based on a given predicate.[31]

```
mapMaybeE :: Event a -> (a -> Maybe b) -> Event b

filterE :: Event a -> (a -> Bool) -> Event a
filterE e f = mapMaybeE (\v -> if f v then Just v else Nothing)

filterE_ :: Event a -> (a -> Bool) -> Event ()
filterE_ e f = filterE e f -=> ()
```

We can also take an Event of `Maybe` valued occurrences, and drop all those that are `Nothing`, using `isJustE`.

```
isJustE :: Event (Maybe a) -> Event a
isJustE = mapMaybeE id
```

As with listeners we can convert an Event with occurrences that contain lists of values into occurrences for each value.

```
mapEs :: Event a -> (a -> [b]) -> Event b
```

---

[31] The Haskell Maybe type represents values that may or may not occur:
`data Maybe a = Just a | Nothing`

```
fromListE :: Event [a] -> Event a
fromListE = mapEs id
```

We can sample the time when an Event occurs using `withTimeE`.

```
withTimeE :: Event a -> Event (a, Time)
withTimeE_ :: Event a -> Event Time
```

We can sample behaviors on Event occurrences using `snapshotE`.

```
snapshotE :: Event a -> Behavior b -> Event (a,b)
snapshotE_ :: Event a -> Behavior b -> Event b
```

To aid in debugging there is `traceE` which prints a String on every Event occurrence.

```
traceE :: Show a => Event a -> String -> Event a
```

A.2.2 The History Based Combinators

All of the Event combinators seen so far are unaffected by an event's *history*. In other words, each Event occurrence is unaffected by those from the past. There are, however, some Event (and Behavior) combinators which depend on an event's history. As a simple example, consider `onceE`. This generates a one shot Event: it generates only one occurrence from its argument Event and then behaves like `neverE`. In Fran this has the type:

```
onceE :: Event a -> Event a
```

The existence of future occurrences clearly depend on the existence of a past occurrence. The important question is when do we start listening for the first (and only) occurrence. Initially, we might assume that the answer to this would be simple. However, imagine that we had an Event which generated left mouse press occurrences (`lbp`). This might generate occurrences from the beginning of the program. Imagine that at some later point in the program we wished to add a listener to print a message based on the next left mouse press. We might try to use `onceE`. However, we would not be interested in all previous event occurrences (the event history), just the latest one. It is sometimes useful to be able to apply a combinator that accesses this history. However, it both leads to an inherent space leak if we hang on to the entire history; and we must be able to access the event occurrences that occurred after a given time.

The two semantic models presented in Section 7.1 grapple with this problem and handle it in different ways. The first assumes that events will maintain their history and that an explicit action must be taken to age the event (and ignore this history). In contrast, the second assumes that events will be aged implicitly and that an explicit action must be taken to hang on to this history. Chapter 7 presents three different implementations which attempt to handle this problem according to the different semantic models. The second and third attempt to implement the two respective models.

The first implementation takes a slightly different approach (discussed in Section 7.2.5), but is closest in spirit to the second model. That is it assumes implicit ageing. All of the Event (and Behavior) combinators that depend on an Event's history are moved into the IO (and GUI) monad. They become actions that respond to event occurrences only once they have been performed. Consider the `onceE` combinator. It now takes the type:

```
onceE :: Event a -> GUI (Event a)
```

When this action is performed it yields a new event which will generate only one occurrence (the first occurrence *after* the action has been performed).

We can ask for only distinct occurrences of an Event using `distinctE`.

```
distinctE :: Eq a => Event a -> GUI (Event a)
```

We can associate the values from an Event stream with the values from a list using `withElemE`.

```
withElemE :: Event a -> [b] -> GUI (Event (a,b))
withElemE_ :: Event a -> [b] -> GUI (Event b)
withElemE_ e bs = withElemE e bs ==> snd
```

We can define a version of scan on events. The function `scanlE f a0 e` yields a new Event that maintains a current internal value, `a`, (with initial value `a0`). On every occurrence of value `b`, it applies `f a b`, to yield a new internal value. The new Event produces this new value as its occurrence value.

```
scanlE :: (a -> b -> a) -> a -> Event b -> GUI (Event a)
```

We can accumulate a value via a function valued Event using `accumE` which can be defined in terms of `scanlE`.

```
accumE :: a -> Event (a -> a) -> GUI (Event a)
accumE a0 e = scanlE (\a f -> f a) a0 e
```

We can generate a tick event that goes off at a given interval using `alarmE`.

```
alarmE :: Time -> GUI (Event ())
```

We can generate an Event level switcher using `switcherE`.

```
switcherE :: Event a -> Event (Event a) -> GUI (Event a)
```

The event level switcher starts by generating occurrences from the first event. After every event-valued occurrence, the switcher loses interest in the old event and generates occurrences from the new event.

If we don't want to lose interest in the first Event we can do this using `manyE`.

```
manyE :: Event a -> Event (Event a) -> GUI (Event a)
```

We can use the event-level switcher to define a monadic-like instance for events. This begins by generating no occurrences (that is it behaves like `neverE`). After every occurrence of `e`, it applies `f`, to create a new event, and generates occurrences from this new event.

```
bindE :: Event a -> (a -> GUI (Event b)) -> GUI (Event b)
bindE e f = neverE `switcherE` (e ==> f)
```

Because some event combinators require to work in the GUI monad it is in fact more useful to have a bind combinator of the following type.

```
bindE :: Event a -> (a -> GUI (Event b)) -> GUI (Event b)
```

This allows a GUI valued function to be applied to the result. For instance, consider the definition of a double click. This takes a timeout value and a click event and generates double click occurrences. After every click it sets an alarm which will tick at a given time. It then merges this alarm with the click. The alarm generates the value Nothing, denoting a failure and the click generates the value Just () denoting a success. We accept only the first occurrence from this combined event, to ensure that we can only either timeout or succeed. Finally, we filter to accept only the successes.

```
doubleclick :: Event () -> GUI (Event ())
doubleclick timeoutval click = do
 click `bindE`
   \_ -> do {e <- alarmE timeoutval;
             fmap isJustE $
                    onceE (e -=> Nothing .|. click -=> Just ())}
```

We can define the `bindE` function by making use of `mapGUIE`. This takes a GUI valued function to apply to each occurrence and an event and by performing a GUI action, returns a new Event which generates the resulting values.

```
mapGUIE :: (a -> GUI b) -> Event a -> GUI (Event b)
```

Using this we can define `bindE`. It first applies `f` to each event occurrence, and then makes a switcher based on this new event.

```
bindE e f = do
  e' <- mapGUIE f e
  neverE `switcherE` e'
```

A.3 What can we really do with Behaviors

We now come to Behaviors. There is one simple built in behavior, `constantB` which is a behavior that always has a single constant value.

```
constantB :: a -> Behavior a
```

Fran supports a second primitive behavior time, which represents the current `time` (specifically the time since the beginning of the program). This works well for animations, where the screen will be refreshed regularly. In GUIs, it is generally useful to have more control over the refresh rate of such a behavior. FranTk therefore supports two GUI valued actions which produce a time valued behavior with a given refresh rate. The first gives the time value since the beginning of the program; the second given the time since the action was performed.

```
timeTick :: Time -> GUI (Behavior Time)
timeTickNow :: Time -> GUI (Behavior Time)
```

To avoid clutter in type signatures involving `Behavior` many types have pre-defined synonyms for their behavioral counterparts. For instance, there is `StringB` for `Behavior String`. A full list is available with the type signature summary at the end of this appendix.

A.3.1 Lifted Behaviors

We say a type or function, which has been raised from the domain of ordinary Haskell values to behaviors is "lifted". For example, a function such as

```
(&&) :: Bool -> Bool -> Bool
```

can be promoted to a corresponding function over behaviors:

```
(&&*) :: BoolB -> BoolB -> BoolB
```

The type `BoolB` is a synonym for `Behavior Bool`; most commonly used types have a behavioral synonym defined in FranTk. The name `&&*` arises from a simple naming convention in Fran: lifted operators are appended with a `*` and lifted vars are appended with `B`.

The renaming required by `&&` can sometimes be avoided using type classes. For example, an instance declaration such as the following

```
instance Num a => Num (Behavior a)
```

allows all of the methods in `Num` to be applied directly to behaviors without renaming. Constant types in the class definition cannot be lifted by such a declaration. In the `Num` instance above, the type of `fromInteger` is

```
fromInteger :: Num a => Integer -> (Behavior a)
```

The argument to `fromInteger` is not lifted - only the result. This allows integer constants to be treated as constant behaviors. While `fromInteger` works in the expected way, other class methods cannot be used. In the declaration

```
instance Ord a => Ord (Behavior a)
```

is not useful since it defines operations such as

```
(>) :: Behavior a -> Behavior a -> Bool
```

Unfortunately, FranTk needs a > function which returns `Behavior Bool` instead of just `Bool`. The `Eq` and `Ord` classes are not lifted using instance declarations. Rather, each method is individually renamed and lifted. These are the lifting functions: they transform a non-behavioral function into its behavioral counterpart:

```
($*)              :: Behavior (a -> b) -> Behavior a -> Behavior b

lift0             :: a -> Behavior a
lift0             =  constantB

lift1             :: (a -> b) -> Behavior a -> Behavior b
lift1 f b1        = lift0 f $* b1

lift2             :: (a -> b -> c) -> Behavior a -> Behavior b
                  -> Behavior c
lift2 f b1 b2     = lift1 f b1 $* b2
...
lift7  ...
```

Using these functions, the definition of `(>*)` is

```
(>*)  = lift2 (>)
```

Many standard Haskell Prelude functions have been lifted in FranTk via overloading. For instance, behaviors are instances of Num, Integral, Fractional, Floating.

There is one important new type class: `Gbehavior`. This provides a behavior based conditional operation.

```
class GBehavior w where
  ifB :: BoolB -> w -> w -> w
```

Its instances include

```
instance GBehavior (Event a)
instance GBehavior (Behavior a)
instance GBehavior Component
instance GBehavior WComponent
```

A.3.2 Reactive Behaviors

Events are used to build *reactive behaviors* which change course in response to events. Reactive behaviors are defined using the `switcherB` function:

```
switcherB :: Behavior a -> Event (Behavior a) -> GUI (Behavior a)
```

The function `switcher b e` starts off behaving like b. On every behavior-valued occurrence of `e`, it switches to (and therefore behaves like) the new behavior. It is important to note that this is a GUI action. This is because the switcher is affected by the switching event's history. The new switcher only notices changes that have occurred after the action has been performed (to generate the new behavior).

We can define two very important behavior based functions on top of these.

A `stepper` is a switcher which generates a behavior from constant pieces.

```
stepper :: a -> Event a -> GUI (Behavior a)
stepper x0 e = switcherB (constantB x0) (e ==> constantB)
```

We can use `stepAccum` to generate a switcher that starts out behaving as `x0` and is updated by the function-valued occurrences of its `change` event.

```
stepAccum :: a -> Event (a -> a) -> GUI (Behavior a)
stepAccum x0 change = do {e <- accumE x0 change;stepper x0 e}
```

For instance, we can define a counting behavior with `stepAccum` that counts the number of Event occurrences since it was created.

```
countB :: Event () -> GUI (Behavior Int)
countB e = 0 `stepAccum` e -=> (+1)
```

The implicit parentheses are around the `-=>` expression, since `` `stepAccum` `` has a lower fixity than `-=>`.

On top of this we have `untilB`. This behaves as `a` until the next occurrence of `e` after which it behaves as the behavior provided by the occurrence.

```
untilB :: Behavior a -> Event (Behavior a) -> GUI (Behavior a)
untilB a e = do {e' <- onceE e;switcherB a e'}
```

A.3.3 Turning behaviors into events

We can turn a behavior into an Event using `toStream`.

```
toStream :: Behavior a -> GUI (Event a)
```

This produces a stream of values every time the behavior changes. This assumes a discrete model of time and behaviors. FranTk provides this by providing only reactive Behaviors. Fran does not as it provides the continuous behavior, `time`.

We can define a predicate function in terms of this that generates an Event every time a Behavior has the value `True`.

```
predicateB :: Behavior Bool -> GUI (Event ())
predicateB b = fmap (flip filterE_ id) $ toStream b
```

A.3.4 Sampling behaviors in the GUI monad

Sometimes it may be necessary to sample a behavior from the GUI monad. At any time a behavior has a given value. It is therefore possible to get its value using `getTime` and `at`. The latter samples a behavior at a given time, the former returns the current time in seconds. These two are in fact the primitives used by the various `withTime` and `snapshot` functions.

```
getTime :: GUI Time
at :: Behavior a -> Time -> GUI a
```

It is important to note here that the time is constant in any step. A step begins whenever some user input is handled. A behavior will therefore not change until immediately after the input Event that updates it. All updates based on the input will be handled, all behaviors will then change and the display will be updated.

A.4 Numeric Types

There are a number of numeric types defined as part of Fran and FranTk. These numeric types and functions are available both a static values and as behaviors.

A.4.1 Basic Numeric Types

All scalar types are essentially the same in Fran and FranTk. Synonyms allow type signatures to contain extra descriptive information such as `Fraction` for values between 0 and 1 but no explicit type conversions are required between the various scalar types.

```
type RealVal  = Double
type Length   = RealVal  -- non-negative
type Radians  = RealVal  -- 0 .. 2pi (when generated)
type Fraction = RealVal  -- 0 to 1 (inclusive)
type Scalar   = Double

type Time     = Double
type DTime    = Time     -- Time deltas, i.e., durations

data Point2     -- 2D point
data Vector2    -- 2D vector
data Transform2 -- 2D transformation

type RealB       = Behavior RealVal
type FractionB   = Behavior Fraction
type RadiansB    = Behavior Radians
type LengthB     = Behavior Length
type TimeB       = Behavior Time
type IntB        = Behavior Int

type Point3B     = Behavior Point3
type Vector3B    = Behavior Vector3
type Transform3B = Behavior Transform3
```

A.4.2 Points and Vectors

Fran and FranTk support vectors and points. These are two distinct types. It is therefore not possible to use + to add a point to a vector. Vectors are a member of the `Num` class (making them numbers) while points are not; thus + works with vectors but not points. Although it is in class `Num`, the * operator cannot be used for vectors.

Read the '.' in the operators below as 'point' and '^' as 'vector'. Thus `.+^` means 'point plus vector'.

```
origin2             :: Point2B
point2XY            :: RealB -> RealB -> Point2B
point2Polar         :: LengthB  -> RadiansB -> Point2B
point2XYCoords      :: Point2B  -> (RealB,RealB)
point2PolarCoords   :: Point2B  -> (RealB,RealB)

distance2           :: Point2B  -> Point2B  -> LengthB
distance2Squared    :: Point2B  -> Point2B  -> LengthB
linearInterpolate2  :: Point2B  -> Point2B  -> RealB -> Point2B
(.+^)               :: Point2B  -> Vector2B -> Point2B
(.-^)               :: Point2B  -> Vector2B -> Point2B
(.-.)               :: Point2B  -> Point2B  -> Vector2B

xVector2, yVector2 :: Vector2B   -- unit vectors
vector2XY          :: RealB -> RealB -> Vector2B
vector2Polar       :: RealB -> RealB -> Vector2B
vector2XYCoords    :: Vector2B -> (RealB, RealB)
vector2PolarCoords :: Vector2B -> (RealB, RealB)
```

```
instance Num Vector2
 -- vectors are a numeric type, however the functions fromInteger, *
 -- may not be used
```

A.4.3 Vector Spaces

The vector type and the scalar numeric types (Float and Double) support several standard mathematical operators. The following operators are therefore allowed on general vector spaces.

```
zeroVector :: VectorSpace v => Behavior v
(*^)       :: VectorSpace v => ScalarB -> Behavior v -> Behavior v
(^/)       :: VectorSpace v => Behavior v -> ScalarB -> Behavior v
(^+^),(^-^):: VectorSpace v => Behavior v -> Behavior v -> Behavior v
dot        :: VectorSpace v => Behavior v -> Behavior v -> ScalarB
magnitude  :: VectorSpace v => Behavior v -> ScalarB
magnitudeSquared :: VectorSpace v => Behavior v -> ScalarB
normalize        :: VectorSpace v => Behavior v -> Behavior v

instance VectorSpace Double
instance VectorSpace Float
instance VectorSpace Vector2
instance VectorSpace Vector3
```

A.4.4 Transformations

The type `Transformation2B` represents geometric transformation on widgets, points, or vectors. The basic transformations are translation, rotation, and scaling. Complex transformations are created by composing basic transformations. The class `Transformable2` contains 2D transformable objects.

```
class Tranformable2B a where
   (*%)    :: Transform2B -> a -> a   -- Applies a transform
```

These are the operations on 2D transforms:

```
identity2  :: Transform2B
translate2 :: Vector2B -> Transform2B
rotate2    :: RealB -> Transform2B
compose2   :: Transform2B -> Transform2B -> Transform2B
inverse2   :: Transform2B -> Transform2B
uscale2    :: RealB -> Transform2B  -- only uniform scaling

move :: Transformable2B a => Vector2B -> a -> a
move dp thing = translate2 dp *% thing

moveXY :: Transformable2B a => RealB -> RealB -> a -> a
moveXY dx dy thing = move (vector2XY dx dy) thing

moveTo :: Transformable2B bv => Point2B -> bv -> bv
moveTo p = move (p .-. origin2)

stretch :: RealB -> ImageB -> ImageB
stretch sc = (uscale2 sc *%) -- 1.0 = 180 degrees

turnLeft, turnRight :: Transformable2B a => FractionB -> a -> a
turnLeft frac im = rotate2 (frac * pi) *% im
turnRight frac = turnLeft (-frac)

instance Transformable2B Point2B
instance Transformable2B Vector2B
instance Transformable2B RectB
```

A transformation that doubles the size of an object and then rotates it 90 degrees would be
`rotate2 (pi/2) 'compose2' uscale2 2.`

Note that the first transform applied is the one on the right, as with Haskell's function composition operator `(.)`.

A.4.5 Fran overloaded functions

Many Prelude functions have been lifted in Fran and FranTk via overloading:

```
(+)         :: Num a => Behavior a -> Behavior a -> Behavior a
(*)         :: Num a => Behavior a -> Behavior a -> Behavior a
negate      :: Num a => Behavior a -> Behavior a
abs         :: Num a => Behavior a -> Behavior a
fromInteger :: Num a => Integer -> Behavior a
fromInt     :: Num a => Int -> Behavior a

quot        :: Integral a => Behavior a -> Behavior a -> Behavior a
rem         :: Integral a => Behavior a -> Behavior a -> Behavior a
div         :: Integral a => Behavior a -> Behavior a -> Behavior a
mod         :: Integral a => Behavior a -> Behavior a -> Behavior a
quotRem     :: Integral a => Behavior a -> Behavior a ->
                             (Behavior a, Behavior a)
divMod      :: Integral a => Behavior a -> Behavior a ->
                             (Behavior a, Behavior a)

fromDouble  :: Fractional a => Double -> Behavior a
fromRational:: Fractional a => Rational -> Behavior a
(/)         :: Fractional a => Behavior a -> Behavior a -> Behavior a

sin         :: Floating a => Behavior a -> Behavior a
cos         :: Floating a => Behavior a -> Behavior a
tan         :: Floating a => Behavior a -> Behavior a
asin        :: Floating a => Behavior a -> Behavior a
acos        :: Floating a => Behavior a -> Behavior a
atan        :: Floating a => Behavior a -> Behavior a
sinh        :: Floating a => Behavior a -> Behavior a
cosh        :: Floating a => Behavior a -> Behavior a
tanh        :: Floating a => Behavior a -> Behavior a
asinh       :: Floating a => Behavior a -> Behavior a
acosh       :: Floating a => Behavior a -> Behavior a
atanh       :: Floating a => Behavior a -> Behavior a
pi          :: Floating a => Behavior a
exp         :: Floating a => Behavior a -> Behavior a
log         :: Floating a => Behavior a -> Behavior a
sqrt        :: Floating a => Behavior a -> Behavior a
(**)        :: Floating a => Behavior a -> Behavior a -> Behavior a
logBase     :: Floating a => Behavior a -> Behavior a -> Behavior a
```

These operations correspond to functions which cannot be overloaded for behaviors. The convention is to use the `B` suffix for single argument functions and a `*` suffix for operators.

```
fromIntegerB    :: Num a => IntegerB -> Behavior a
toRationalB     :: Real a => Behavior a -> Behavior Rational
toIntegerB      :: Integral a => Behavior a -> IntegerB
evenB, oddB     :: Integral a => Behavior a -> BoolB
toIntB          :: Integral a => Behavior a -> IntB
properFractionB :: (RealFrac a, Integral b) =>
                           Behavior a -> Behavior (b,a)
truncateB     :: (RealFrac a, Integral b) => Behavior a -> Behavior b
roundB        :: (RealFrac a, Integral b) => Behavior a -> Behavior b
ceilingB      :: (RealFrac a, Integral b) => Behavior a -> Behavior b
floorB        :: (RealFrac a, Integral b) => Behavior a -> Behavior b
(^*)          :: (Num a, Integral b) =>
                           Behavior a -> Behavior b -> Behavior a
(^^*)         :: (Fractional a, Integral b) =>
                           Behavior a -> Behavior b -> Behavior a
(==*)         :: Eq a => Behavior a -> Behavior a -> BoolB
```

```
(/=*)          :: Eq a => Behavior a -> Behavior a -> BoolB
(<*)           :: Ord a => Behavior a -> Behavior a -> BoolB
(<=*)          :: Ord a => Behavior a -> Behavior a -> BoolB
(>=*)          :: Ord a => Behavior a -> Behavior a -> BoolB
(>*)           :: Ord a => Behavior a -> Behavior a -> BoolB
notB           :: BoolB -> BoolB
(&&*)          :: BoolB -> BoolB -> BoolB
(||*)          :: BoolB -> BoolB -> BoolB
pairB          :: Behavior a -> Behavior b -> Behavior (a,b)
fstB           :: Behavior (a,b) -> Behavior a
sndB           :: Behavior (a,b) -> Behavior b
pairBSplit     :: Behavior (a,b) -> (Behavior a, Behavior b)
showB          :: (Show a) => Behavior a -> Behavior String
```

Appendix B Usability Evaluation for Rapid Prototyping

B.1 Introduction

This thesis is about prototyping tools for developing interactive systems. The purpose of prototyping is to support iterative design and so provide systems that can be evaluated. It is therefore useful to consider the range of techniques available for performing usability evaluations. There are a variety of techniques available to perform such evaluations ranging from quantitative[196] to qualitative approaches[203], and from analytical techniques[142], through structured lab experiments [175] to fully situated field trials [11]. The choice of which technique to use depends on the purpose of the evaluation, and what sort of system is being evaluated. This appendix discusses briefly the various types of evaluation and suggests what might be achieved with each. It highlights those that require software prototypes, and outlines what the technique requires from a prototype. It then selects a set of techniques appropriate for use with rapid prototyping, in the early stages of iterative, multi-user, interactive systems development. It goes on to discuss the evaluation of the multi-user, design rationale editor developed as the QOC case study (see Chapter 2), using the selected set of approaches. It outlines the objectives and setup of the evaluation, the results of the evaluation and conclusions about the methods used.

B.2 Usability Evaluation Methods

There are a great variety of techniques for performing evaluations. The Usability Now programme of the DTI [126] recognised a number of evaluation methods:

- **Analytical evaluation** – the use of theoretically based, analytical methods to look in detail at a design. These approaches include psychologically based models such as GOMS, Cognitive Complexity Theory and SANE and the use of formal methods in HCI [37] (see Chapter 8). These approaches tend to focus on quite detailed aspects of a design such as menus or dialogue design.
- **Expert evaluation** – the use of expert knowledge in evaluation interfaces. These approaches include cognitive walkthrough[121] and heuristic evaluation[142].
- **Survey evaluation** – these approaches involve using questionnaires and interviews to find out information from real users about a design. Questionnaires can ask users to write about specific questions in detail, or can provide sliding scales where users provide values to get details such as satisfaction with a system.
- **Experimental evaluation** – these are focused studies attempting to answer specific design questions. They are performed with real users. For instance, we could perform an experiment to find which of two sets of icons were more recognisable.
- **Observational evaluation** – real users can be observed performing task of varying complexity. These observations can take place in a lab setting or in the field. They can be performed with both fully operational software systems and simple paper prototypes. They can be used to provide both qualitative data, such as what sort of problems users had, and quantitative data, such as how long it took a user to perform a particular task. At their most sophisticated these studies may involve long term studies in the field, making use of ethnographic techniques to gather data.

Evaluations can be designed to be both formative and summative. Formative evaluations are used to feed into designs and help guide them. Summative evaluations are used to try to validate that a system is really usable and meets its requirements. Most of the above approaches can be used for either purpose. With summative evaluations, we must, however, be very careful. Their scope must be broad enough to really try to both verify that a system meets a set of requirements, and validate that it is actually of use. Complex field trials, that test a system in its final place of use, may be necessary before the latter can really be claimed. This appendix concentrates on the goal of providing formative information for redesign, as this is the focus of evaluation for rapid prototyping.

B.3 Low and High Fidelity Prototypes

Evaluations can be performed with both paper based and fully implemented prototypes. The former involve the use of low-fidelity prototypes that can cover a variety of issues. For instance, cardboard models of computers can be used to show the size and layout of hardware and to provide details about the size of buttons that people might want. Sketches can show what an interface could look like. Users can step through interaction by selecting objects on a sheet and having the evaluator place the next sheet in front of them. Anecdotal evidence suggests that rough images can encourage people to talk about fundamental issues, while more polished interfaces make people worry about low-level, "look" issues[113]. Paper prototypes can also be modified by users and designers allowing co-operative redesign [17]. However, people often have difficulty understanding exactly what interaction will really be like. It will not be possible to discover subtle or complex usability problems if the user is not able to actually use a system for real. There is therefore still an important need for software prototypes early in the design process.

B.4 Analytical Evaluation

There are a number of ways of going about analytical evaluations. They generally rely on some formal model of the system and some abstract notion of usability or model of the user's activity. There are a number of approaches based around cognitive models of the user's activity. These approaches tend to provide very low level information that can be used to think about specific design questions; for instance, an analysis of the number of keystrokes involved in a set of different designs. These approaches do not rely upon having access to a working system prototype. They are therefore not directly relevant to the prototyping work in this thesis. As they do not involve users they are dependent on design assumptions, and are fairly restrictive. They are best used in addition to other more user-centred styles of usability testing.

B.5 Expert Evaluation

There are a number of approaches to expert evaluations: the two most commonly referenced are the cognitive walkthrough[121] and heuristic evaluation[142].

B.5.1 Cognitive Walkthrough

The Cognitive Walkthrough is an approach that attempts to simulate the interaction involved in performing a task. It may be performed with a model of a system [36] or with the system itself [42]. It was originally designed to evaluate simple "walkup and use" systems. It is based on a theory of exploratory learning, and some guidelines. Its theoretical basis therefore requires a good knowledge of psychology such as goal structures and activation of goals. However, it can be used by designers without this sort of psychological background. A study by John[98] found that "little or no experience in either user interface evaluation or cognitive psychology is required of the user [of the CW technique.]"

The approach can, however, be very time consuming[167]. For instance, John [98] found that an evaluator took 16 hours to perform a walkthrough of just over 100 user actions. The Cognitive Jogthrough [167] was developed to deal with this problem. This involved a group walkthrough using a version of Rieman's Walkthrough evaluation sheet. The discussion was moderated to keep it on track, and recorded to prevent the need to wait for a reporter. In a ninety minute walkthrough about 30 user actions were discussed. The group still found it too time consuming to evaluate all core tasks with this approach. These methods would tend to be used with a design specification; however, they can be used with actual software prototypes. Again they rely on the design group's intuitions being correct. As discussed in Chapter 1 this may not always be the case, especially with complex systems such as Air Traffic Control systems.

B.5.2 Heuristic Evaluation

Heuristic evaluation[142] involves having a group of evaluators look at an interface and judge it against a set of recognised principles, as shown in Figure 51. It is less formal than structured walkthrough methods and so was intended to be cost effective. In one study Nielsen[142] found that it could pick up between 20 and 60% of problems found in actual use. It can also generate "false positives", that is problems that are never experienced in real use. It can be carried out with a system specification or a

real prototype. With the specification, it can be applied earlier; however, with a real prototype it will frequently be more effective.

• Use a simple and natural dialog	• Provide clearly marked exits
• Provide an intuitive visual layout	• Provide shortcuts
• Speak the user's language	• Provide good help
• Minimise the user's memory load	• Allow user customisation
• Be consistent	• Minimise the use and effects of modes
• Provide feedback	• Support input device continuity

Figure 51 - Heuristic Evaluation Guidelines

There have been a number of comparative studies of heuristic evaluation and cognitive walkthrough [99][36][42]. These studies have shown that the two techniques are complimentary. Heuristic evaluation tends to find higher level problems, while cognitive walkthrough finds lower-level problems. In one experiment, applied to a real prototype the evaluators found that both techniques took about the same amount of time, and they found no clear difference in the number of problems found between both methods [42]. Judgements about the severity of problems are not well supported by either technique [98]. The effectiveness of these techniques is therefore very limited. Studies have tended to concentrate on systems designed for novice users. Though they can be useful, the lack of real user involvement can be problematic. User-centred evaluation is still necessary.

B.6 User Based Evaluation

The alternative to expert and analytical approaches are techniques that involve real users. There are a number of ways of going about these. However, their effectiveness centres on being able to involve a representative set of participants from the end-user population. This involvement should help to overcome the considerable knowledge differences that can exist between design teams and the consumers that a system is intended for.

While iterative development and user involvement can be vital to the development of good applications, they are not always possible [80]. When developing "off the shelf" software for general groups of users it may be difficult to gain access to a truly representative set of participants. Even when developing for more specific groups of users, there may be problems. Management at customer sites may not see the benefits of taking users away from their regular work. Usability issues have to fit into the broader area of product development. For instance, marketing groups may see themselves as the only legitimate group who should have contact with customers and may be worried that developers will alarm customers, or have a negative effect on the product's image. Competitively bid contracts may involve one group developing requirements, while another performs the design. User involvement may therefore be difficult or impossible in the late stages of product development. However, the difficulties and importance of developing a good interface to a system, make it important that users are involved in the design.

There are a number of important questions to answer when performing a user based evaluation:
- Where to perform it?
- How to get users to interact with the design?
- How to gather data?
- How to analyse data?
- How to redesign?

B.6.1 Where to Perform It?

Evaluations can take place in some form of lab, or at the user's place of work (in the field).

B.6.1.1 Lab Testing

Lab studies can be very useful for certain kinds of testing. For instance, Philips[205] favour them for focus groups sessions in early development; prototype testing that can't be done in context; the study of

novice users exploring an interface; comparative testing; fine tuning near the end of a design. Usability labs themselves usually contain recording equipment, such as video cameras and scan converters. The latter takes images directly from the VGA output of a computer. These provide a good means of gathering data. Labs may also have large one-way mirrors, allowing a group of people to watch the evaluation progress. This may let design teams and management view the test. Viewing problems with a design as they occur is a far more effective way of conveying them than simply reading about them [205]. When evaluating some software, lab tests may be the only possible approach. For instance, it is clearly not possible to evaluate a prototype Air Traffic Control system in real use. Here realistic, lab based simulations are the only option.

B.6.1.2 Field Studies

Where possible, field studies can be a far more effective way of evaluating systems [205][167]. The situated nature of work is very important [191]. Suchman argues that the way people carry out their work is deeply situated in the context of their work. Any evaluation that takes place in a different context could result in very different results. Zirkler et al [213] discuss the approach taken by an electronic publishing company. They found that the work environment plays a critical role in determining how customers use their products. Customers changed their use of the software significantly in the lab as opposed to in their place of work. In the field, customer were able to refer to their own work samples during evaluations. For busy customers, carrying out any evaluation at their place of work may also be more efficient. It should be noted that there are still significant differences between demonstrating a piece of software at a user's place of work, and actually carrying out a realistic evaluation in the field. Prototypes for demonstration purposes must be mobile, for instance, this could be done with a laptop. Prototypes for situated field trials must actually integrate with systems in the user's place of work.

B.6.2 How To Get Users To Interact With Design?

Designs can be presented to users through a variety of techniques. These include demonstrations, directed dialogues, by having users perform a set of basic given tasks, allowing free use to do some actual real work, or by carrying out an actual situated study of the system in the users place of work.

B.6.2.1 Demonstrations

A system can simply be demonstrated to a user, using for instance a live demo or a video recording. This sort of activity can give users a general idea of what a system will be like, and allow some feedback. Demonstrations can be used to focus later evaluations; allow a larger group of users an understanding of the system; and give users a chance to start thinking about a design [213]. A prototype for such an evaluation need only implement the necessary functionality to run the demonstration; it does need to be ready early in the design process. Usability testing of prototypes with users can get more reliable data when performing specific tasks than when just demonstrating an interface. Users may come to rapidly different subjective opinions, especially if problems are glossed over [196] [114].

B.6.2.2 Directed Dialogues

A more user involved form of demonstration is a directed dialogue [150]. The customer is directed, by a tutor, through a task, which is to be performed with the simulated system. If the customer has difficulty, the tutor guides them to screen areas to help them on. This progressively discloses information to the user until they are able to understand how to perform the task. This can be done with one or more users[131]. The focus is on seeking feedback from users about specific tasks with a given design. It can be used with paper based prototypes, as the user is being shown a system and asked about their understanding of it, rather than having to use it themselves. However, it can also benefit from the use of a software prototype. Again the prototype need only implement the minimal set of user interface level features necessary. For instance, it need only simulate the minimal necessary amount of underlying application behavior.

B.6.2.3 Evaluation with simple tasks

Users can be given a prototype of a system and be asked to carry out a set of very specific tasks (e.g. [212]). These may be at the level of finding a specific bit of data from a database, or making some specific changes to a paragraph in a text editor. Scenario design is very important here, as the tasks tested need to be realistic. If the scenarios are badly designed the evaluation could be testing irrelevant details. This sort of evaluation can help test very specific aspects of a system, and see whether users really are capable of using and understanding it. They test user understanding better than

demonstrations or directed dialogues because the user is in control. This sort of evaluation will only verify that a system is usable for a specific set of tasks, it will not validate that the system will really support the user in their day to day work. High level misunderstandings about the nature of requirements will not be caught. This approach can be used to test a partial prototype of a system, as only the features necessary for the specific evaluation need to be implemented [211]. The implementation of these tasks must, however, be more complete than for demonstrations or directed dialogues.

B.6.2.4 Evaluation with free use

We can choose to evaluate a system by allowing users free use of the system. In this case they will be given a high level goal and be allowed to proceed in using the system to carry out this goal. The choice of high-level task is again important here. It can be useful to get users to try out some realistic work with a system [203]. For instance, we might get them to write a short paper with a word processor. This kind of evaluation takes more time. It is semi-situated, in that users are performing realistic tasks with a system. The complexity of the task determines how long the evaluation should take. A good free evaluation might unravel over an extended period of time. It might be used as a longitudinal study, to determine how users both learn to use a system, and then work with it when they become skilled [130]. Where possible, this form of evaluation may be carried out in the users workplace. This could be particularly important if the environment in which a user works has an effect upon their use of a system. Such cases might, for instance, involve high background noise, or low light level.

In order to support such evaluation, a prototype must provide a realistic implementation of a system. It needs to implement all of the user interface features required for such use. It does not, however, need to implement all underlying application behavior. For instance, the prototype does not need to talk to the real back end systems, such as existing data bases, that would exist in real use. These can instead be simulated. As another example, a prototype Air Traffic Control system does not need to communicate with real planes; and does not necessarily need to be implemented as a distributed system. It only needs to simulate the behaviour of aircraft sufficiently for a controller to be able to interact with them.

B.6.2.5 Situated Evaluation

The most complex style of evaluation is a situated evaluation that tests a version of the system in the users own workplace, where it is used for real work. This form of evaluation will find a whole host of problems that result from the complex nature of the workplace. Lab tests can provide some information, but to truly understand a system, users need to be able to try out software in their own environments for reasonable periods of time. Situated studies look at how well the system fits into the workplace, not simply how well it fits the requirements *as understood by the design team.* This style of evaluation becomes more important when evaluating multi-user systems, as discussed in Section B.7.

Beta testing can be used to find usability problems as well as simply looking for bugs and performance problem. Smilowitz et al [186] compared lab, forum and beta tests as a way of performing usability testing. In the beta tests, participants were able to download the software and try it out in their real work environments. They were asked to observe and record problems in their use of the software tool. Such tests make data capture more difficult. They found that beta testing found the same number of problems as lab testing, though it found a lower percentage of severe problems. In such cases, understanding the exact nature of the users problems depends heavily on their ability to record them accurately.

A situated evaluation or beta testing requires a complete – or at least almost complete – implementation of a system. It must be possible to perform real work, rather than simply simulating it. It need not, and almost certainly will not, implement all features of the proposed system. However, the core features that it does implement must be fully operational. For instance, it must have real underlying application support to handle data from existing databases.

B.6.2.6 Prototyping and Evaluation in this thesis

The prototypes that were developed in this thesis were aimed at the first four types of evaluation. The systems could be demonstrated to the user or designer, or be evaluated with simple tasks. For instance, this took place with the ATC case study. A Human Factors expert performed specific tasks with the ATC system, and discussed desired redesigns. The usability evaluation carried out with the QOC editor was more complex, requiring free use of the system. The development of systems for situated

evaluation is a more difficult issue, as these require more realistic implementations. Such evaluations are therefore unsuitable for the early stages of iterative development.

B.6.3 How To Gather Data?

There are a variety of evaluation techniques that can be used to gather data. These will be discussed briefly, as we made use of a number of techniques in the QOC evaluation, discussed in Section B.9. These include:

- concurrent verbal protocols, or "think-alouds" where a user talks through what they are doing;
- retrospective verbal protocols, where users go back through what they have done and discuss it;
- system logs, that record user interaction at the keystroke level;
- surveys such as interviews and questionnaires;
- and video and audio recordings of user interaction.
-

B.6.3.1 Concurrent verbal protocols

One of the simplest and most popular approaches to gathering data during an evaluation is to use a "Think-aloud" protocol [212][114][109]. Here a user is requested to talk through what they are doing as they are doing it. This allows an understanding of what is really happening when problems occur. Nielsen argues that it "may be the single most valuable engineering method" [143]. It can, however, be difficult to get users to explain what they are doing without sufficient practice. It may feel unnatural and interfere with the task that they are performing [213].

One approach that tries to overcome this problem is co-operative evaluation[212]. Here the evaluation becomes a conversation between the participant and the person running the evaluation. Users are asked to consider themselves as co-evaluators, and are asked questions such as "What will the system do if ...?". A study with a group of software engineers showed that this method could be effective, even with very little training. An alternative is to use a co-discovery approach[196]. Here a group of users work together to perform a task and therefore discussion should ensue that results in them naturally thinking aloud. Participants need to be equally matched for this to work; otherwise, it may turn into a lesson where one participant attempts to teach another. This also mirrors how new users may learn to use a new system; learning is often done in groups [167]. Molich [131] found that performing a think-aloud with a group of three to five users should find somewhere between 50% and 90% of problems. Unlike expert evaluations, these approaches do not generate "false positives"; all problems are faced by real users in practice. These approaches are therefore fairly cheap and effective ways of carrying out usability evaluation.

B.6.3.2 Retrospective verbal protocols

To avoid interrupting the user during an evaluation, a retrospective verbal protocol can be used. In this case, the user is taken back through their earlier interaction and questioned about it. The replay of earlier events can take place via videotapes (e.g. [192]) or by having users retype the keystrokes that they previously performed from recordings in a system log [211]. Users may reinterpret their previous actions incorrectly; however, this can be an effective way of getting data. Henderson [89] found that concurrent and retrospective verbal protocols tend to find the same set of problems. Both concurrent and retrospective verbal protocols are centred around user discussions and so require no explicit tool support themselves. Because of the need for some recording of the user's activity, retrospective verbal protocols are usually more time consuming than concurrent ones.

B.6.3.3 Software logging

Software logging requires explicit tool support. For instance, the Microsoft Usability lab has developed logging tools that handle time stamping and storage of user interaction[93]. Their tools locate episodes based on which events denote the beginning and end of the episode. Different user intentions may produce the same event. Events must therefore be examined in the context of the surrounding data. Event data can be filtered, and keystrokes can be compressed, for instance, into words that the user typed. Playback tools can then be used to replay the results of event logging software, but they only work when the playback computer is in exactly the same state as the original computer.

Software logs provide a detailed record of what the user did. They are amenable to automatic searches, and quantitative analysis. The mass of data gathered can, however, be intimidating. Analysis can therefore be time consuming. More importantly, they do not provide an understanding of why a user

behaved in the way that they did. They must therefore be used in addition to verbal protocols and survey approaches.

FranTk currently provides no explicit support for software logging. Future work could investigate how to integrate this into the toolkit.

B.6.3.4 Survey approaches

To compliment data gathered by direct observation, survey approaches are often used [51][63]. Debriefing sessions can be used. These may be less formal than retrospective verbal protocols, simply discussing the user's views about the system. Pre and post evaluation questionnaires can also be used [167]. Pre-evaluation questionnaires can find background about users that may affect their performance during the evaluation. Post-evaluation questionnaires allow users to note complaints, or grade the system.

Questionnaires also exist to attempt to fathom user performance. For instance, the NASA developed Task Load Index (TLX), is an internationally recognised approach to data gathering [14]. It attempts to assess a user's subjective workload. It uses six scales that cover factors that influence experience of workload. These cover areas such as mental and physical demand, time pressure, frustration, effort and performance. Each is measured on a sliding scale, and is compared pairwise with the other factors for importance. This captures factors that may be very important but is still relatively easy to carry out.

B.6.3.5 Video and audio recording

Video and audio recording are very popular ways of gathering data during an evaluation (e.g. [11][123][130] [150][205]). Audio data can be used separately and transcribed using conversation analysis techniques (e.g. [135]). This makes it far easier to study in detail and search through. This can be used in conjunction with system logs to provide an understanding of the context of the user activity [93]. Together audio and video recordings can be very effective capturing subtle user activity. Several cameras can be employed to capture what happens on screen, users' expressions and gestures. Video recording can also be an effective way to "sell the results" to management and designers [205].

Video analysis can have problems. While recording can capture most activity; it cannot always capture everything. Video data seen out of context can be misconstrued [124]. Involving participants in the analysis can make up for this [192]. Video can simply be used as the focus for discussion between designers and participants. It is also important to minimise distractions when performing an evaluation [175]. Participants can be easily intimidated by recording equipment. However, researchers at Philips have found that participants soon forget about well placed video equipment [205].

Video analysis has a reputation for being time consuming [123]. It can be cumbersome to edit, and difficult to locate information. This can be made even more difficult if several cameras are used during evaluation. Editing equipment can help here by merging video streams onto a single tape. This style of equipment was used in the QOC evaluation discussed in this appendix. A large observation team may offset the need for video recording [196]. However, such a team may be significantly more off-putting for a user. Video analysis tools can help. For instance, the ability to tag particular events on a tape and then provide random access can be very powerful [123]. Tool that allow system logs to be combined with video tapes can also be very powerful. The level of analysis required becomes an important issue here. Video recordings can be used simply to elicit retrospective discussions with participants, and help recall some key episodes of the interaction. This style of analysis requires much less time, effort and software support. It is, however, less effective than more detailed tool supported analysis.

B.6.4 How To Analyse Data?

There are a variety of techniques that can be used to analyse the data gathered during an evaluation. These approaches fall into two camps: quantitative and qualitative methods. Quantitative approaches look for measures and values that can be considered and compared; qualitative approaches look for detailed process information, the "how and why" of what actually happened.

B.6.4.1 Quantitative Approaches

One of the most significant features of quantitative data is that it can be used in a cost-benefit analysis. It can be used to determine how effective a new interface may be; for instance, timing information can

show why a new interface will save money [196]. It can, however, be very expensive to perform such evaluations. BT estimated that to perform an evaluation using 20 participants on 5 tasks cost £1500 to recruit participants, £500 of material, 10 person days to plan, 30 days to undertake the trial, and 30 days to analyse the results[63]. Trials of this sort are not conducive to rapid, user-centred, iterative development.

Considerably cheaper quantitative studies can be carried out, however. Szczur et al [196] discuss a low cost quantitative approach to evaluation based on usability metrics. Evaluations involved 1 hour sessions that tested 12 categories with 45 individual tasks. The usability metrics suggested include time-to-learn, memory retention, number of errors and user perception. These can give details of how successfully users perform the designed tasks. Computer logs, audio tapes and direct observation can be used for quantitative measurements. Attitudes based on interviews and questionnaires can be gathered. To make such usability tests amenable to statistical decision making requires a significant number of participants. This significantly raises their cost. Expert judgements are still required to determine when a system satisfies the usability criteria.

Statistical techniques exist for analysing quantities of data. For instance, tools such as SHAPA and MRP exist that can be used to look for cycles of behaviour that reoccur. Cuomo [34] used such tools to find a commonly repeating pattern in user interaction. Future design work could then be applied to increase the efficiency with which users can perform such actions. However, the key to such analysis is interpreting the statistics. We can only use such data if we also understand what the user is actually doing. Qualitative data is therefore necessary to complete such analyses.

Quantitative approaches can be very powerful for comparative studies. These allow a measure of success to be defined. For instance, we could compare a specific aspect of two designs based on timing criteria, or workload measures such as the NASA TLX. Significantly such experiments should be replicable. Experimental set-up is very important here. Designs must be compared with equivalent evaluators, and with a large enough number to ensure statistically relevant results. Between-group and within-group experiments are possible [141]. In the former, two separate groups of participants are used. In this case care must be taken to match each group's experience. In the latter setup, the same participants can be used in each experiment. In this case, groups must be counter balanced. Half the group should use one design first; the other half should use the other. To analyse such results, we must assume that the effects of learning are symmetrical. Comparative statistical methods can be used to analyse the results such as t-tests or ANOVAs [51]. Different techniques are required to compare within and between subject groups [141]. These sorts of experiments can only be used to answer small scale design questions. When performing large, complex tasks, there are too many dependent variables. Users may carry out a task in one of so many ways. The effect of a participants experience also becomes a very serious issue. The situated nature of work also becomes important. A particular design may perform well under lab conditions, but suffer significantly in real world conditions. Ensuring ecological validity is therefore a significant problem.

B.6.4.2 Qualitative Approaches

The quantitative approaches discussed above can be used to compare designs, and to find if problems exist, such as whether specific tasks are performed too slowly. However, they cannot be used to find what the actual problems are, why they occurred and how they can be solved. Qualitative data is needed to find out how to go about redesign. The objective in qualitative analysis is different from quantitative experiments. Wright and Monk [211] argue that though results still need to be as generalisable as possible, the time scale involved in rapid prototyping is far too short to include an evaluation method that would be considered scientifically, statistically valid. Formative evaluation is not about validating a design; it is about finding as many problems as possible. Generality is only really significant if changes are suggested that are very expensive or seem counter-productive. Twidale argues that "All problems with a system scale up and out: any success may not"[203]. Interface features which feel cumbersome in small tests will almost certainly irritate in real use.

We can look for different types of problems in evaluation data. For instance, we can look for critical incidents. These refer to cases where a non-optimal path has been taken by the user when performing some task. These commonly occur when a user makes an error and has to recover from it. They also occur when a user has to perform more actions than is strictly necessary, such as might occur when having difficulty finding a menu item. This sort of analysis requires an understanding of the task that a

user is performing and probably the strategy that they are using to perform it [211]. Finding critical incidents can therefore be difficult. The notion of optimal paths becomes even more difficult in learning or exploratory phases of interface use.

An easier form of analysis is to look for breakdowns. Winograd and Floris[210] argue that a system is usable when it allows tasks to be carried out in a "transparent way". When the computer becomes the central focus of the user rather than the task in hand, then a "breakdown" has occurred. In these cases, we look for cases when the user has to think about how to fix a problem with the interface, for instance, when it becomes excessively awkward to use. Finding breakdowns requires verbalisations from the user. A concurrent verbal protocol should be used so that we can tell when the user is facing problems. Wright and Monk[211] found that breakdowns are easier to find than critical incidents, as they involve an obvious complaint by the user. They argue that "most critical incidents are accompanied by a breakdown but ... the converse is not true". Breakdowns also tend to result from more critical problems in a design.

There are a number of more complex approaches to evaluating qualitative data such as distributed cognition and ethnomethodology. I will return to these when discussing multi-user evaluation, in section B.7.3, as they have been developed to apply to environments involving groups of users.

B.6.5 How To Do Redesign?

Once we have data from an evaluation, working out how to use it in redesign is important. Often the insights gained from think-aloud style evaluations are enough to work out what problems a user is having, and to carry out redesign. However, care must be taken in redesigning properly, or the benefits of evaluation may be missed. Too little design analysis may result in numerous iterations, without removing significant design defects [207].

Structured approaches exist that attempt to guide redesign. Sutcliffe et al [194], for instance, show how model matching can be used to consider usability problems. They compare hierarchical task-action models of what the user tried to do with system models. These are used to guide redesign. For these to work, a high-level system specification must exist. We must be careful with such approaches. They can encourage overly restrictive systems that support simply what users did under one set of circumstances, rather than supporting their more general high level goals.

An alternative approach to this is to involve users more thoroughly in cooperative design [112]. For instance, technological redesign can involve a playschool session where groups of users discuss the system and try out various redesigns with paper and pencil sketches [166]. Wilson et al [209] discuss a case study that tried to encourage user involvement in the design. Two users co-operatively developed task models, and redesigned paper and pencil prototypes. They argue that such approaches can be powerful, but need to be carried out carefully. Users must not be expected to be full designers themselves. This may place too great a burden on them. If only a small number of users are involved, they may be unrepresentative of the general user population, especially if they have become heavily immersed in a particular design.

B.7 Evaluating Multi-User Systems

The evaluation approaches discussed so far have concentrated on techniques for evaluating single user systems. There are a host of additional problems that occur when evaluating multi-user systems.

B.7.1 Problems With Evaluating Co-Operative Systems

The evaluation of co-operative systems requires a more complex and widespread evaluation approach. Grudin [81] has identified a number of serious problems that can occur because computer support for groups of people has a number of different characteristics from that designed simply for single users. There have been a number of major groupware failures that have resulted from this situation. For instance, when those who do the real work are different from those who benefit from the system, there may be no incentive to use a system.

The situated nature of work is particularly significant. Social and political aspects of a workplace are important[81]. For instance, meeting systems have failed because users were unwilling to admit that particular meetings were of low priority. Design argumentation software has failed because it does not

support non-rational decision making. For instance, one such system failed because the manager wanted his group to project a strong sense of consensus. Groupware systems require to be adopted by a critical mass of users to be worthwhile. Lab tests can fail to bring out basic practical problems. For instance, co-authoring software which looked great in the lab has failed because users have been unwilling to give up their favourite authoring software. A highly-motivate group can make even an awful group application work; a badly managed installation can destroy even the best of products. The actual procedures which people use to carry out their work are far more complex than those used in handbooks. They are usually far more flexible. High-level organisational rules may vanish at the group level. A simple evaluation of basic workplace tasks will therefore be invalid.

Determining what problems exist within an organisation can also be difficult. For instance, in one project involving engineers and architects, delays were occurring[165]. The manager thought it might be lack of communication and installed video conferencing software. Unfortunately, things became worse. Engineers were over-committing themselves to attempt to keep up good working relationships with the architects. With more communication this problem simply became more problematic.

B.7.2 Possible Approaches

There have been some attempts to develop specific methodologies for evaluating co-operative systems. For instance, Ross et al[166] argue that evaluating CSCW systems requires a number of techniques. Evaluation is required of both the collaborative activity through a theoretically driven evaluator's perspective, and of the tool itself through a practical participant's perspective. Evaluating the technology is a process of verification. It demonstrates that the software meets the design requirements and has no major usability problems. It should be carried out early on, and such evaluation can take place in the lab. New technology will change how work is carried out in an organisation. Evaluating the activity is a higher level problem and is more sensitive to the situation in which the system will be used. It is a process of validation; validating that the overall redesign of the work situation is actually effective and suits the needs of the organisation as a whole. This sort of validation needs to take place in the field. The issue of multiple stakeholders is particularly important here. For instance, a system can make the work for a group of employees less interesting while providing useful information for management. Evaluations must be sensitive to these differing needs.

B.7.3 Methods of Analysis

There are a number of relatively light-weight approaches that can be used to analyse data from a multi-user evaluation. They include looking for conflicts in the use of a shared system, breakdowns or looking for process dependent variables. They are designed for evaluations involving "free use" (see Section B.6.2.4). They do not, however, demand that an evaluation be carried out in a fully situated, work-place study. They can therefore be used to evaluate the tool itself in an initial study.

B.7.3.1 Conflict Detection

Morris et al[135] discuss the evaluation of a shared windowing system. They used lab based experiments with a mixture of observational, survey and experimental evaluation to remove gross errors. They used software logging to capture user interactions. They used conflict detection with the log to highlight where two users tried to act on the same object. As these could be difficult to interpret from the logs, the context of use, including audio transcripts and video logs, was compared to find out exactly what was happening. This sort of approach focuses attention on low-level details about user interaction.

B.7.3.2 Breakdown analysis

The notion of breakdown analysis, initially discussed in section B.6.4.2 has been extended by Scrivener et al [180]. They distinguish a number of different classifications of breakdown.

1. User and tool – breakdowns can occur with actual use of the software;
2. User and task – breakdowns can occur when a user does not understand the task in hand, or does not have the necessary knowledge to carry it out;
3. User and environment – breakdowns can occur with the environment that the user is working in;
4. User and user – breakdowns can occur in communication between different users, if several users are working cooperatively with some system.

Analysis of user-tool breakdowns is equivalent to that discussed for single-user evaluation. Analysis of user-user breakdowns can cover low level problems caused by poor design of the tool. For instance, in a

synchronous groupware product the tool may make it difficult to understand what another user is doing. Alternatively, the breakdown in communication may be caused by poor design of the collaborative activity itself. Breakdowns with the task are problems with the collaborative activity. Finally, breakdowns with the environment result from problems in the context and environment in which a user is working.

B.7.3.3 Finding process dependent variables

Monk & McCarthy[132] discuss a number of factors that they used to evaluate a text communication system. They looked at process dependent variables that could be affected by the alternative communication medium. People will go along way to protect their primary task, for example, the actual topic under negotiation and so secondary tasks, such as communication style itself, may suffer. They used transcripts to look at concepts such as utterance and interruptions, task focus and the need for explicit topic openings. This approach gave them more significant results than more crude measures of task performance. This approach is a qualitative analysis mechanism that can be used with data gathered from lab tests or situated trials.

B.7.3.4 Multi-user heuristic

Rodgers[165] adapts some concepts from Green's work that should be considered when evaluating multi-user systems: distributed knock-on viscosity and gradient of resistance. The former refers to extra activities that some users must perform to get a system to work. The latter refers to problems making changes to a prototype or system. For instance, the more established a working practice, the greater the resistance to change.

Ross et al [166] found a number of general categories to be important in the results of an evaluation of a shared text editor. These heuristics mirror some, and expand on other, single-user usability heuristics as discussed in section B.5.2. These new heuristics include:

- feedback and awareness – are users aware of what they and others are doing;
- focus – are users aware of change without being overwhelmed;
- coordination – how do users get to perform activities;
- ownership and control – are users happy about who owns what, who has precedence or permission to change data;
- communication – is both peripheral and explicit communication well supported in the group. The former refers to communication that occurs as a side-effect of something else, but still serves to enhance awareness in the system;
- mental models and metaphors – how do users deal with notion of self + task + computer + group + group task.
- consistency – do metaphors convey the "beingness" and awareness in a system. The lack of standards in groupware systems makes this less of a problem but there is a need to be consistent with single user systems.

B.7.4 Heavier Weight Analysis Methods

There are a number of more complex theoretical techniques that can be used to gather detailed information from situated field trials. These include conversation analysis, interaction analysis, distributed cognition and ethnomethodology.

B.7.4.1 Distributed cognition

The understanding that actions are situated and that group work relies on social factors has led to a change in the cognitive science community. Theories such as distributed cognition have been developed. The main goal is to consider cognition as a group activity, which takes place through the "propagation of representational state across media". Such media can include external objects such as computer and paper based displays. For instance, when studying navigation on a ship, rather than looking at individual members of a team and their tasks, it looks at how distributed activities are carried out, the way knowledge is transmitted between team members and the artefacts that are used for this purpose. It involves detailed video analysis to look for subtle incidents that take place and underlie group activity. Gestures and glances can be very important in this. It has been applied to look at Air Traffic Control, software engineering and hospital wards [83][165]. It can be a very expensive technique to use requiring very detailed analysis of small sections of video tape.

B.7.4.2 Interaction analysis

Suchmann and Trigg [192] present interaction analysis. This is the study of the interaction of people with each other and with the material world. They look for categories of different types of interaction, and evidence for each. They used three different recording approaches: setting-oriented - looking at a particular location; person-oriented – following an individual around; and object-oriented – tracking a particular piece of technology or other artefact (such as a paper record). This approach is designed to consider situated activity and again makes intensive use of video analysis.

B.7.4.3 Conversation analysis

Conversation analysis can provide detailed information about interaction between participants in a study. For instance, people co-ordinate on content and find common ground for communication in a process called grounding. Grounding [28] is the collective process participants use to try to reach a mutual belief. Acknowledgements and back-channel responses (e.g. "uh-huh") are important to this process.

There are a number of constraints on grounding that can be affected in multi-user systems:

1. co-presence – being in the same physical environment;
2. visibility – the ability to see each other;
3. audibility – the ability to hear each other clearly;
4. cotemporality – whether communication is synchronous or asynchronous (as happens with e-mail);
5. simultaneity – can both communicate at once;
6. sequentiality – can communications get out of order;
7. reviewability – can a user review messages sent by the other person, for instance, with a written message;
8. revisability – can a user revise messages before sending them, allowing repair to be done privately rather than publicly.

These constraints can affect how easy users find communication and how easily they can work together in a collaborative system. Conversation analysis involves studying transcripts to investigate how communication and co-ordination are being affected by the new system. Again this makes it a heavy-weight method. However, a basic understanding of grounding and the constraints upon it can be used as heuristics or as guidelines when carrying out a more light-weight study.

B.7.4.4 Ethnomethodology

The need to study work in context, and therefore to perform situated evaluations, has led to the adoption of ethnographic methods within the CSCW community. Lucy Suchman used ethnomethodological approaches as the basis of her "Plans and Situated Actions" [191]. This inspired much of the work in this area. She argues that human activity is richly context based; it can be impossible to remove human activity from its context. There are two basic premises: "first, that what traditional behavioral sciences take to be cognitive phenomena have an essential relationship to a collaboratively organised world of artefacts and actions, and secondly, that the significance of artefacts and actions, and the methods by which their significance is conveyed, have an essential relationship to their particular, concrete circumstances".

Hughes et al [95] have attempted to show how ethnography can be used in design. They suggest several approaches.

- **Concurrent ethnography** – where design is influence by an on-going ethnographic study while developing a system. They tried out the approach with an air-traffic control system going through the cycle of fieldwork>prototype iteration>fieldwork about four times. The team was small enough to allow an informal transfer of information between ethnographers and designers.
- **Quick and dirty ethnography** – brief studies are undertaken to provide a general but informed sense of the setting. This was carried out in an industrial environment, on a larger project. The study was able to influence the design, focusing on informing strategic decision making. The team found diminishing returns with extended trials.
- **Evaluative ethnography** – verify or validate a set of already formulated design decisions. This was used as a sanity checking method, to evaluate a model of work in a financial company. Ethnography was used to tweak the systems.

- **Re-examination of previous studies** – previous studies can be re-examined to inform initial design thinking. This is a cheaper method as it attempts to reuse existing data. However, it may be dangerous if the context of work is too different.

Ethnographic methods face problems [95]. There are problems of scale. It is relatively easy to study small scale confined environments. Scaling up to environments with work distributed in time and space can be difficult. Secondly, time pressures are very important. Social research can last years. This sort of time-scale is totally impractical for design. The ethnomethodological approach insists on a rigorously descriptive rather than theoretical program. This produces rich descriptions of work in context, but creates problems when trying to produce models and ideas about design. These rich descriptions can be difficult to communicate to designers as they are typically lengthy and discursive. The introduction of technology to support large-scale activities can transform small scale ones, thereby undermining the large ones. This has been described as the "paradox of system design" [20]. Ethnographic methods are therefore not a ready made solution. Decisions about how to use data from ethnographic studies are still difficult. However, these methods can provide very rich data which can be very useful if carefully integrated with design.

B.8 Summarising Evaluation Techniques

This appendix has highlighted a number of techniques for evaluating interactive system. These have covered both single and multi-user systems. This section will summarise these and highlight those for which the prototypes developed in this thesis are suitable.

As we have seen evaluations can take a number of forms. The variation falls along a number of axes.

- What aspect to evaluate – tool or activity;
- What scale of evaluation to carry out – from demos, through simple task based evaluations and free use, to fully situated evaluations;
- Where to carry out the evaluation – lab or field;
- What style of prototype to use – from low fidelity paper prototypes, through initial software prototypes, to full-scale implementations;
- What sort of analysis – quantitative v qualitative analysis.

These axes are not orthogonal; not all combinations make sense. The first axis provides the important dividing line.

B.8.1 Evaluation of the Activity

The more complex of the two aspects is evaluating the activity. This involves evaluating the new work practices which introduction of the new technology will produce. Such evaluations must take place in a realistic setting; this usually means a fully situated field trial. Such evaluations do not necessarily require a prototype system; they can be used to evaluate a model of activity being used for design. In this case they can be used in the early requirements gathering phase. An example of this is the use of "evaluative ethnography" to test a set of design decisions. This sort of trial will also feed into smaller scale, tool based evaluations, as the activity model will be used to guide design of tasks for lab trials. To really understand what will happen when a new system is introduced, we must evaluate the tool and the new work practices together. This must be done with a realistic prototype. Such studies are not always possible in the field. For instance, it is clearly not possible to test a prototype Air Traffic Control system with real work. Such studies must take place in realistic setups. For instance, NATS uses full scale simulation suites that accurately replicate the environment in which controllers work. These sort of situated (or semi-situated) studies are generally beyond the scope of early prototypes of the form developed in this thesis.

B.8.2 Evaluation of the Tool

The prototypes developed in this thesis are more suitable for tool centred evaluations. Such studies must be intelligently designed, making use of realistic tasks. These studies clearly also involve the evaluation of low-level aspects of the activity or individual tasks, such as how users perform basic co-ordination with a tool. They do not, however, concern themselves with a global view of the activity as a whole.

Tool centred evaluations can take a number of forms, from early demonstrations with either paper or software prototypes, through to "free use" with an initial software tool. These studies can take place in the lab or at the user's place of work. However, in the latter case they are different from fully situated trials, as they will still not take place in the full working environment. They may seek different styles of data:

- early feedback on a design – this can use demonstrations, and involves gathering simple qualitative data;
- finding major usability problems – while this can be done with directed dialogues, it generally involves use of an actual software prototype. Again this involves gathering qualitative data;
- fine tuning – evaluations can seek to find and tune important activities. For instance, we might wish to speed up an important interaction technique. This generally involves gathering both qualitative and quantitative data.

When carrying out early iterative design it is important to recall that "All problems with a system scale up and out: any success may not"[203]. Such evaluations should concentrate on finding these usability problems. The exact techniques chosen depend on the system being evaluated. We can evaluate either single-user or multi-user aspects of a CSCW system. For instance, it would be less relevant and more difficult to evaluate in single-user mode a tool that was designed only for communication, such as a text based talk system.

There are a number of general techniques suitable for this style of evaluation.

- Scale – Such evaluations can involve anything from demonstrations to free use;
- Data gathering – gathering data via verbal protocols, such as "think-alouds" or "co-discovery" is probably the most effective way. Audio and video data can also be gathered, and used with retrospective verbal protocols.
- Data analysis – Analysis of gross errors such as breakdowns or conflicts in a multi-user system. Video analysis can also be used. While analysis can be very heavy-weight, this is only necessary if seeking subtle data. If looking for gross errors such as breakdowns, then analysis can be more lightweight.

B.9 A Case Study - The QOC Evaluation

B.9.1 The System to be evaluated

The use of rapid evaluation techniques will now be demonstrated through an evaluation case study of the QOC multi-user design rational editor. The editor was designed for the Questions, Options, Criteria (QOC) notation. The interface was introduced in Chapter 2. Each user has a separate view of the QOC collection. Each QOC is maintained in a window. Different users can have windows open in different areas of the screen. Within each window, however, users views are strictly WYSIWIS (What-You-See-Is-What-I-See). The level of sharing is important. Users can see changes made by anyone in any of the other visible windows. They can also see where the other users' cursors appear in their own window. Users have a colour associated with them, used for their cursor. A different colour scheme is used to represent changing options. Objects being edited by a user appear in green; objects being edited appear in red. Locking is at node level so two users can both act in the same window but cannot both act on the same node. The prototype was initially developed in the Clock language (see Section 3.12). It was this version which was evaluated. A version was later developed in FranTk.

This study had several meta-level purposes.

- It demonstrates that the prototype developed was sufficiently efficient for real evaluation. It is important to note that while the Clock version was evaluated, the FranTk implementation was more efficient and so would have been adequate for this purpose.
- It provided some real redesign that had to be done to the prototype. The changes made to the Clock prototype were later made to the FranTk system.
- It demonstrates that the system developed was capable of carrying out an actual task.
- It provides an interesting case study in evaluation itself, and provided some interesting results about use of a shared editor.

B.9.2 The Study

B.9.2.1 The Task

Several groups of users worked together to summarise a section of an accident report using the QOC notation. There were four pairs and one group of three users.

Users were first given a chance to learn to use the system. This involved trying out the interface by cooperatively producing a given QOC. The users then went onto the main task. They were provided with several pages of the London Ambulance Service (LAS) report and given a brief overview of what was involved in the area that they were discussing. The section in question contained details on why the LAS might want a computer aided dispatch system, whether they might go for an existing system and who they could buy one from. The participants were asked to analyse and summarise the decisions made in this restricted section of the report through the use of a set of QOCs. The whole task took just over an hour for each group.

The choice of task was important. The study was concerned with the tool itself. This included how well it supported the basic task of producing and modifying QOCs, and how well it supported co-ordination and co-operation through the various shared awareness mechanisms. This meant that we required a task which involved discussion, and in which the focus of the users should be on gathering and summarising data; breakdowns would therefore occur when the focus of the user became the tool itself rather than the task in hand. However, the task had to be sufficiently consistent so that each group would at least use the same data, even if they gathered and handled it differently. The goal was to summarise a design rational from a specific document. This meant that the each group should have had the same general style of discussion, as they were working with the same basic information. This kept the task consistent between groups of users. The evaluation was also able to provide feedback about the different mechanisms that each group used to summarise the data. However, it remains an open question whether such a scenario would actually be realistic in any given work environment.

B.9.2.2 The physical setup

The evaluation was set up to look at distributed use of the system. Even in such a setup audio communication between participants would be vital. The participants therefore needed to be placed so that they could not see each other, but could hear each other. Participant could have been placed in separate rooms and have communicated over a telephone. However, it was felt that it would be logistically simpler to place them in a single room separated by a partition. This would result in similar interaction properties, but made it easier to monitor what each of the participants was doing. The full setup can be seen below.

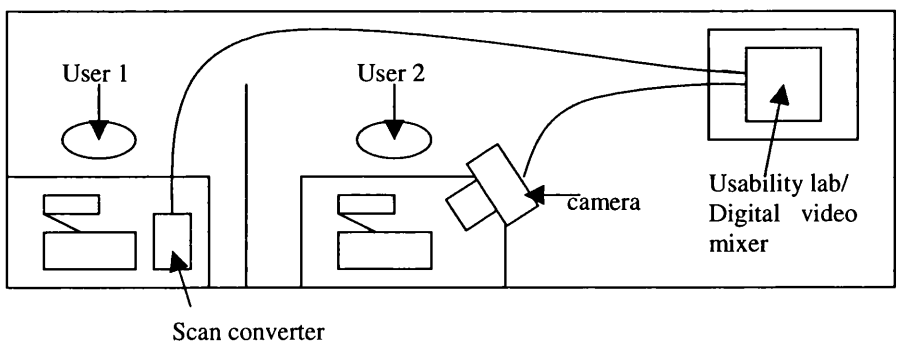

Figure 52 - Setup for Experiment

B.9.2.3 Information gathering

I made use of a think-aloud, co-discovery protocol. The existence of multiple users testing the same piece of software made this very simple. Participants discussed their work. They also discussed problems with each other during the evaluation. For instance, they had to learn co-operatively how to use the system. This worked very naturally. The first reaction of any participant to a problem was to ask their partner about it. This is unsurprising as most users when working in company tend to learn by discussion rather than by reading user manuals [167]. As the participants were in the same room I could

monitor what both were doing. In the case of a major breakdown between the user and tool I could interfere and switch to a co-operative evaluation style, allowing the user to elaborate on their problem and so to fully understand it.

The participants were also recorded with video cameras and a scan converter to record the output from the screen. To be specific, one user was recorded with a camera and one with a scan converter. Users were given radio microphones. The recording was carried out with a portable usability lab. This allowed the output from the video recordings to be combined onto one screen, and composed the input from the radio microphones. The two images were scaled and placed one on top of the other. This produced an image that while no longer fully readable, was still good enough to provide an understanding of the general user interaction. When evaluating with three participants we were able to record only two of the users' screens, along with audio from each user.

I carried out interviews with the participants at the end of each session. The participants were therefore able to express their subjective opinions about use of the system. As advised in Fowler et al[63], I did not show video tapes to participants immediately after the evaluation sessions, but instead discussed problems noted on paper. This proved effective enough for discussing the major breakdowns that were noticed during the evaluation itself; the problems were generally memorable and the prototype itself could be used to highlight points in the discussion. While the use of video tape might have added to the discussion, the logistics of finding appropriate points on the tape for discussion would have been cumbersome and forced unacceptable pauses in the discussion. The users were somewhat tired and were certainly not in a position to immediately spend another hour studying a videotape. Users were also given the NASA TLX questionnaire to attempt to highlight which areas of workload were most significant in the evaluation.

B.9.2.4 Data analysis

During the evaluation I was able to pick up major breakdowns or User Interface Disasters as Molich [131] terms them. After the evaluation, video analysis was carried out. Here I was able to look for more subtle problems than could be caught during the evaluation itself. The analysis focused on looking for breakdowns, critical incidents where possible and problems in each of the multi-user heuristic categories suggested by Ross [166]. I also looked for more general patterns in the way each of the groups worked. Due to timing and logistical constraints, I was unable to show and discuss the videos with each set of users. In retrospect this was a bad idea. The number and scale of the problems caught in the actual evaluation sessions was deceptively impressive. There were a number of minor breakdowns that were caught only during the video analysis. More significantly, the search for general patterns of behaviour was only really possible using the video tapes. The audio conversation helped to convey what the user was doing. However, it was not always perfect at doing this. A better evaluation setup would have involved video analysis immediately after the evaluation, with further discussion with participants later in the day.

B.9.2.5 Redesign

During the usability evaluation I used what has been termed a progressive usability format [109]. This meant that major usability problems were fixed after each test session, before the next group of participants used the system. This was useful because it meant that participants in each session did not spend their time struggling with the same basic problems found in the previous session. Instead, they were able to work on and find new issues. Large usability problems often mask smaller ones that might otherwise be found [51].

B.9.3 The Evaluation Results

A number of problems were raised relating to both single and multi-user issues. Some of these were simple ones that should have been caught by basic modelling or heuristics. Others, however, were more serious high-level issues that could only really be considered as part of a user evaluation.

B.9.3.1 Single User Problems

During the evaluation we found a host of problems that applied to single user issues in the design. Heuristic evaluation and more detailed modelling could have caught a basic set of these problems. For instance, we can categorise a number of these problems as violating task conformance, visibility or predictability.

A number of problems crept into the prototype as a result of inadequately thought out redesigns. These problems were usually very low level issues. In the pilot study users complained that it took too long to add new nodes (they had to go through the menu system). A toolbar was added to allow new questions, options and criteria to be created more quickly. Unfortunately, not enough feedback was provided about the generation process and there was no way to stop creating a node. These two problems are violations of visibility and reachability properties, and so should have been caught by heuristics: not enough data was provided about the state of the interaction (we're trying to create a question) and not every state can be reached (want to stop creation part way through).

A check for task conformance in the specification could also have prevented another simple problem. (That is checking to see if the system supports each task required.) A node has both a type (Q,O,C) and a name. Initially, there was only one menu command "Edit node" that allowed both to be edited. Both users took several attempts to find how to edit a name, while one assumed that it was impossible to edit the type of the node, and so accomplished a change by deleting and recreating. This is a clear example of a critical incident where a user followed a non-optimal path to achieve a goal.

Some more complex issues arose relating to single user issues. An imperfect understanding of the users' task caused further confusion. A QOC may have consequent QOCs which follow on from an option. Several users concluded from this that to create a consequent question, they should use the pop-up menu available from a current option. Instead the design model assumed that a new QOC would first be created and that they would later be connected. A better understanding of the users' task was required to solve this.

A greater understanding of the strategies used to create QOCs was also gained during the evaluation. Users were looking through a body of data and using QOCs to summarise and analyse the data. The general strategy appeared to be: (1) read through a section and work out what it was about; (2) work out the "Question" being discussed; (3) choose options corresponding to those available; (4) select relevant criteria. Several users first added a set of boxes for each criteria that they would need; then they named them; finally, they connected them to options. This creation strategy therefore followed a very modal approach: Create Question, Create Options, Create Criteria, connect them. In the interview section, two of the users noted that they found the mechanism of "click to create, click to place" frustrating. Given the modal strategy it might therefore have been appropriate to have used a modal interface as happens in normal drawing tools. Though some of this analysis came from the post evaluation interviews, the video analysis proved very valuable in finding supporting evidence for this hypothesis.

Finally, several users felt that the QOC notation as it stood was inadequate for the task in hand. Nodes could be annotated with page and section numbers to relate objects to sections of the report. However, a greater flexibility in appearance was requested to distinguish between concepts that were explicit in the report; implicit in the report (and inferred by the analyser); and concepts missing from the report, but worthy of note. This sort of classification could refer to both options and criteria, and to judgements about whether options fulfilled criteria, how well they fulfilled them and how important these criteria were.

B.9.3.2 Multi-user issues

There were a number of multi-user issues raised during the evaluation that referred to both interaction level notions such as shared awareness and higher level issues about the way that users went about working together. While I had rough intuitions about some of these issues immediately after the evaluation, the video analysis was very important for these discoveries, particularly those involving use of the shared cursor.

I found the categories of issues (Feedback and Awareness; Focus; Coordination; Ownership and Control; Communication; Mental Models and Metaphors; and Consistency) raised in [166] to be useful here. It was also helpful to try to understand shared interaction by looking at the objects used to carry out that shared interaction.

Feedback and Awareness

Provision of feedback is very important in groupware applications, as users must be aware of both what they themselves and the other participants are all doing. Shared awareness gives "an understanding of others, which provides a context for your own activity" [39]. Users need to be able to answer questions

about "Where am I?", "What am I doing?", "Where are you?", "What are you doing?" and "Who did that?" [130].

Shared awareness was provided in two ways in the system. Users could see changes being made by others in their current window. Any object being changed appeared in red. Secondly, there were shared cursors which showed the position of other users in a window relative way. This could, however, be confusing when a cursor vanished from a window. Several breakdowns were observed, where users explicitly asked "where have you gone?". Providing highlighted information about which window each user was in may have helped here.

This shared awareness information was used intensively. The shared cursors were particularly helpful. Having a general awareness of where the other user was seemed to be important to most of the participants. However, there were different opinions about how much shared information was important. Two participants felt that knowing exactly what their partner was up to was very important. This extended to the point where users wished to be able to see pop-up menus being selected by their partner. In the absence of this extra information, the shared cursor was used to provide this information. One participant could watch their partner's cursor and by its position infer what exactly they were doing. In contrast, several other participants were far less concerned with this information, and felt providing more information would just be noise. This shows the need for tailorable awareness information. A simple solution would have been to provide text labels for objects being changed by others, explaining what was happening to them. Activation of these labels could then be user controlled.

Understanding who was doing what proved easy in the two person tests. The shared cursors and green/red colour scheme proved effective once users got used to them; adjustment was very rapid. However in the three person test understanding was much more difficult. This is unsurprising as the red/green colour scheme was designed for pairs. There were two main coping strategies. Firstly, the shared cursors were again used to provide information about who was over a particular object and so who was editing it. Secondly, a list of users currently editing in a window appeared in the top right of that window. This information was far more heavily used. However, as it appeared in a different place on the screen it proved difficult to get at.

The importance of understanding "who was doing what" was heavily task dependent. This was particularly noticeable in the three person trial. In the initial warm up task where all three users were working simultaneously to try to create a single QOC, they became rapidly confused. There was very little understanding of who was doing what. However, when they moved on to the main task these problems reduced to relatively usable level. Discussion and negotiation became more important; it was rare for all three users to be modifying the same QOC simultaneously. This may well have been a coping strategy as a result of earlier problems. However, this still demonstrates the importance of considering low level usability problems in the context of a realistic task.

Focus

In systems with multiple users there may be an overwhelming number of actions happening at once. Noticing what has changed recently or attracting attention can be difficult. In our study, however, these problems were not significant. The size of groups meant that two users could talk to attract each others attention without interrupting others work. This would have been far more problematic if we had scaled up to four users. Focusing other users attention on a particular object on the screen was also not a serious problem. Users tended to say where to look, referring to an item by name. Gesturing with their cursor was then used as a more specific focusing mechanism. An awareness of what their partner was doing was important here.

Coordination & Control

Coordination activity to decide who is going to do what, and who can do what, is very important in multi-user activities. There are different levels of this: from general work organisation to low level interaction.

The use of explicit system based roles have been suggested[11] to control who can do what. For instance, a system could have editor and reviewer roles, where one changes a document and others can comment but not modify. However, several studies have found that roles tend to change dynamically during group interaction [135][130]. Dourish and Bellotti [39] argue that shared awareness can be used

instead; rather than explicitly defining roles and controlling who can do what, if users can tell what is happening then they can negotiate their behavior dynamically. We also found that roles tended to be varied and dynamic. There were a number of different cooperative styles used during the evaluation. The styles used mirrored those found in group writing: scribe/consultant writing (where one users writes suggestions made by the other), parallel writing (where users write simultaneously but separately) and joint writing (where users actually write together).

The most common style of editing in our study was a joint writing style. All groups at some point used a very synchronous collaboration style, where often two people would make changes to different parts of the same QOC. This sort of activity would usually take place after a discussion about what needed changed. A less common variation on this was when two users negotiated by changing. For instance deciding what a question should be by both attempting to change it until they were happy. It took a little time for some groups to scale up to this sort of activity. For instance, one group started off using a turn taking style to make an initial QOC before they became happy enough to try out joint creation. At this stage a scribe/consultant role was also sometimes used. For instance, after finding some criteria in a document, one user would read them out while another added them in. However, this style was far less common.

There were different levels of parallel writing. For instance, one user could look at options for a particular question, while another looked for criteria. Truly parallel writing, working on separate QOCs was far less common. Two of the pairs never tried using this parallel writing style. They just collaboratively negotiated their way through each QOC. In contrast, one of the pairs did spend most of their time using this style of work. They initially negotiated the general structure. Then after negotiating the first question and set of options, one of them went on to add criteria while the other went on to the consequent question. In the three person group, a mixture of these styles could be seen at any one time. Two of the team spent much of their time negotiating one set of criteria, while the other member started looking at a second separate QOC.

In the joint writing sessions, coordination was very important. Locking was at object level, so when an object was in use no one else could alter it. Users could achieve a very fine grain of interleaving actions. For instance, one user added criteria while another connected them up to options. Even this level of locking proved too coarse though. Having the ability to connect between objects that another user was editing proved important. This was the only collaborative activity that two users wished to be able to perform on the same object.

Communication

In collaborative activity communication is very important. It can be explicit or peripheral[166]. The latter is where communication occurs as a side-effect of something else, as opposed to direct discussion. In this study, voice communication was very important. Users would not have been able to proceed without the ability to talk to each other. The problems of private and public communication were not very important here, as there were not enough participants in any group to allow more than one conversation. Morris [135] found that for groups of four people, users could still successfully arrange their own social protocols for use of a single voice channel.

This category relates closely to feedback and awareness as a greater awareness meant a need for less explicit communication. For instance when one pair of users were working separately on different QOCs they needed to occasionally keep track of what their partner was doing. One user, who was particularly concerned about knowing what her partner was doing, kept her windows set up so that she could see what he was doing at all times. In contrast, her partner would explicitly ask how she was getting on. Supporting these various approaches to communication was therefore important.

The main form of peripheral communication was awareness of the other user's shared cursor. Though in all groups users seemed to be using it for general awareness information, there was a distinct lack of any references to shared cursor locations in conversation. None of the users made use of phrases such as "look here"; references were always to explicit objects. However, all but one user did gesture with their cursor, moving and pointing it at objects that they were talking about. Users would also follow each other's cursors about. For instance, one user taught her partner how to connect between nodes in two different windows, gesturing with her cursor; her partner performed the actual actions mirroring her movements with his cursor. This lack of explicit verbal telepointer references has been mirrored in

other studies. In one study, Mitchell [130] found that telepointers were often seen as limited as they could be difficult to draw attention to. One user suggested that they should be able to use drawing as a method of focusing attention. He would have liked to be able to temporarily circle objects for example. This more explicit and noticeable behaviour may have made communication easier.

Ownership
When several different people are creating a document, the issue of who owns what, and who can change it is important. The issue of ownership of objects did arise in this study, especially in the three person group. The level of concern depended on how closely users were working together. Where users were working collaboratively on the same QOC using a joint writing style, there was less concern. However, unless an object had been created after explicit negotiation, participants tended to at least announce their intention to change an object. For instance, with comments such as "I'm messing with your criteria". During one session in particular, there was considerable disagreement between two participants about the naming of several criteria. To handle this problem, rather than argue by changing, participants made their feelings known by adding textual comments to objects (through the pop-up post it note system). When users were working on separate QOCs this notion of ownership became more important. For instance, one user finished his QOC and then moved to his partners. He first explicitly asked permission to help, but still felt unsure about adding extra data: "You know what those criteria are for so you should probably just connect them".

The power to make any desired change was noted several times in the three person group. One user noted that "this is a socialist program", as each user had the same rights and nobody truly owned anything. One of the other users noted that she felt that they'd been a bit too polite when they first started using the system, as they negotiated very carefully what they could change.

Other issues
Issues of mental models & metaphors and consistency were less important in this study. None of the participants had any really significant experience with groupware applications; most had none at all. The issue of previous experience did arise once. One group had some experience of turn taking systems and so assumed that locking would be at window level. This assumption continued to control their behaviour, even when they discovered that they could make certain changes simultaneously. Getting over the notion of window level awareness with object level locking was therefore more difficult than anticipated. The same pair also felt uncomfortable about saving when their partner was working, even though the system allowed it.

B.9.3.3 Conclusions about the case study
The co-discovery, think aloud procedure proved to be an effective way of finding many basic problems in the system. All single user problems were found in this manner. Most breakdowns in multi-user activity were also visible at this stage. However, developing a proper understanding of user activity, particularly in the case of shared interaction really required basic video analysis. A fairly lightweight analysis of the video tape (2-3 times through each tape, with pauses at appropriate points) provided a basic insight into the strategies used by each group. Most of these strategies were again visible during the actual evaluation and interview. However, to find supporting evidence it was necessary to make use of the video tapes. The use of shared cursors was the major issue which became much clearer as a consequence of looking at the video footage. To truly understand what was going on, a retrospective discussion with the users and the video tapes would have been helpful.

B.10 Conclusions

This appendix has discussed a range of techniques for carrying out single and multi-user evaluations. It has highlighted the difference between tool centred evaluations, concentrating on low-level interaction issues, and activity centred evaluations which try to evaluate the tool and new work practices as a whole. It is important to note that both evaluation styles are concerned with the mechanisms used to carry out a set of tasks. Tool centred evaluations, such as the case study discussed in this appendix, are still concerned with how a user, or group of users achieves their goal. More complex situated studies, must consider these issues and consider what other factors affect the general activity.

The prototypes developed with FranTk in this thesis are more suitable for tool centred evaluations. I have concentrated, in particular, on qualitative evaluation mechanisms that attempt to gather data about a users problems, rather than demonstrating some result through quantitative data. Such qualitative studies are more appropriate for the evaluations in the early stages of rapid, iterative development. In particular, I demonstrated that a mixture of co-discovery think aloud procedures and lightweight video analysis were easy to perform and were effective for finding a large number of usability problems, and developing a general understanding of single and shared user activity.

Appendix C Combining Interactors and Haggis

Part IV of this Thesis described one approach to combining formal specifications of interactive systems with high level prototypes. This Appendix discusses an earlier attempt to provide links between these two areas. One possible combination of notations and implementation languages, is to link the LOTOS interactor model to Haggis[32]. There is a strong link between the two systems. Both support concurrent programming, with event based message passing between components. Haggis provides a high level language with efficient support for structured graphics, and interactive components. LOTOS provides a set of powerful temporal operators to allow components to be composed. It also provides reasoning power, which is necessary to aid programmers to understand the concurrent interaction in the system. We can therefore develop programs in a structured way, at a high level of abstraction.

To show how this can be done, we use, as an example, a highly interactive game (the space fighter game from Chapter 2). We specify it in LOTOS, and convert this into Haggis code. Our example shows the power of the Haggis framework: the Haggis representation is at almost the same level of abstraction as the specification. This example is relevant as it demonstrates that Haggis can deal with the kind of highly interactive systems that declarative languages are frequently bad at handling.

C.1 Functional Programming & Executable Specifications

Previous work by Alexander [5], has shown the strong link between formal specifications and functional programming. Alexander used eventCSP, a subset of the concurrent specification language CSP, to specify human-computer interaction [5]. For instance, consider a Quit button, that may appear to be pressed or unpressed, with an updateable label. We could specify it as follows:

```
quitB =  (mouse-down-in-button-> pressedButton ->
          (  (mouse-up-in-button -> quit -> exit)
          [] (mouse-up-outside-button -> quitB)))
       [] (setLabel -> upDateLabel -> quitB)
```

Alexander argued that it is easy to understand and reason about the interaction in this specification. Possible sequences can be built using the prefix (->) operator. Different paths are shown using the choice ([]) operator.

Alexander used an executable subset of VDM (Vienna Development Model) to describe the data, and operations used in eventCSP. This language, called "me too", used pre and post conditions to describe the events [5]. She used a functional language to execute these specifications. We develop her approach, by using newer developments in functional languages, which make executing specifications easier.

C.2 The Example

In the past, Haggis has been used to build relatively simple systems [62]. As a more complex example, we describe the development of a highly interactive, real-time user interface. It is a space ship game, as shown in Figure 53. The user inputs commands via the keyboard. A number of enemy ships fly in waves across the screen. These destroy the player's ship if they collide with it. The player must avoid hitting the hills at the bottom, and the enemy ships. The aim is for the player to destroy the enemy base when it is finally reached, while shooting as many enemy ships as possible. There are buttons to allow the user to pause the game, restart it, or quit from it. The current score will be displayed on the screen. Though the graphics are only simple, the game requires real-time animation. This example therefore provides a highly concurrent system, with a number of different interactors: the game and all the buttons. The specification concentrates on this concurrency.

[32] This chapter is based on a paper that appears in the proceedings of DSVIS[170].

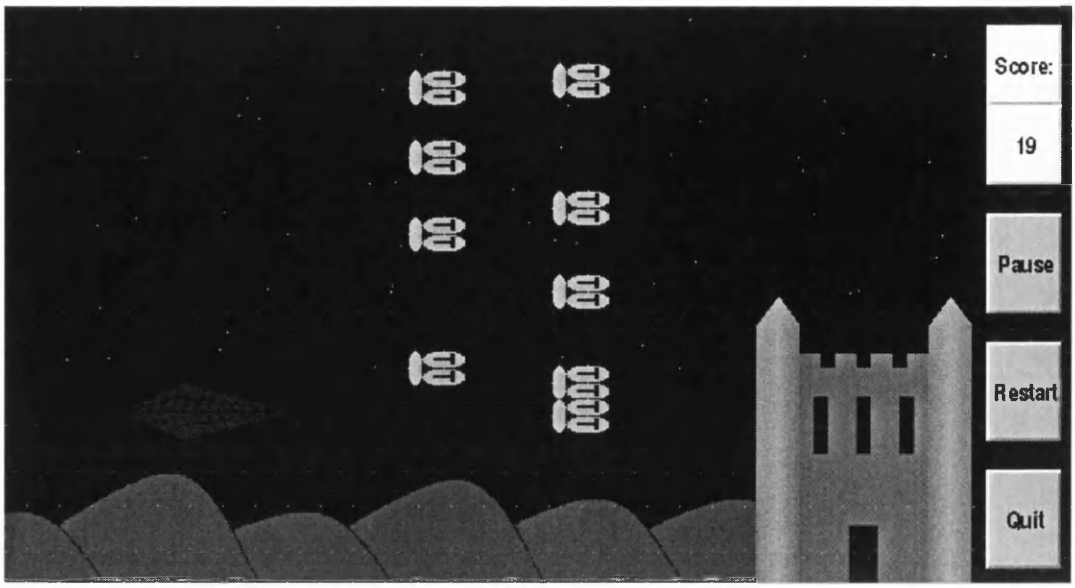

Figure 53 - The Interactive Game in Haggis

C.3 LOTOS Specification

In common with the CNUCE approach to interactors [151], we specify our system in LOTOS.

C.3.1 The Structure of the Game

The game can be modelled as six interacting processes:

- an interactor for the Quit Button (*Quit*)
- an interactor for the Restart Button (*Restart*)
- an interactor for the Pause Button (*Pause*)
- an interactor for the Game Input and Output (*GameIO*)
- an output interactor to display the score (*Score*)
- an application process for the Game itself (*App*)

These can then be synchronised over a number of events to allow the necessary communication. The two processes Main and Game are used to compose the components above, and define how they interact.

```
process Main[nscore,sendpause,
          display,input,
          restart]:noexit:=
 hide oc,is,it,iw in
  ((GameIO[oc,display,sendpause,input,is,it,iw]  |[oc,is,it,iw]| App[it,iw,is,oc,nscore])
  [> (Restart[restart]
   >>  Main[nscore,sendpause, display,input,restart]))
endproc

process Game[display,input,restart,
          pause,quit]:exit:=
 hide nscore,sendpause in
  ((Main[nscore,sendpause,display, input,restart]  |[nscore,sendpause]|
    (Score[nscore] ||| Pause[sendpause,pause]))
    [> Quit[quit]
endProc
```

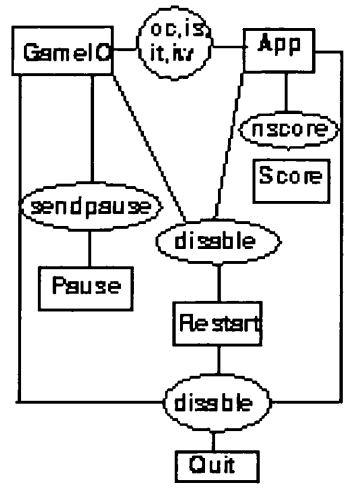

Figure 54 - The Inter-process Communication in the Game

In LOTOS, *a* |[*c*]| *b* means run process *a* in parallel with process *b*, synchronising over gate (event) *c*; *a* ||| *b* means run processes *a* and *b* in parallel without any synchronising; and *a* [> *b* means run process *a* until *b* starts. A process definition *P[a]*, means process *P* with gate *a*; *hide a in P* means that event *a* is visible only in process *P*.

Our specification can be described informally as follows. The Game continues to run, with input being passed from *GameIO* to *App* and output from *App* to *GameIO* and *Score*, until the *Restart* interactor fires (i.e. the Restart button is pressed). At this point *Main* starts all over again. When the *Pause* interactor is fired, the game will be paused, and will remain so until unpaused. This all continues until the *Quit* interactor is fired.

Standard LOTOS does not provide a suspend/resume operator. This operator would make it easier to implement the pause button. A suspend/resume operator has been added to the new LOTOS standard, E-LOTOS [97]. In E-LOTOS, *a* |> *b* (suspend/resume operator) means run process *a* until *b* starts, when *b* finishes, continue with *a*. No tool support currently exists for E-LOTOS so this new operator is not supported. I have therefore not used it in this example.

In the diagram above we can see a graphical representation of the communication in the system. In this Process Interaction Network, processes are represented by named boxes, communication gates by circles, and communication by lines [50]. From this diagram and LOTOS specification we therefore have a clear understanding of how the system fits together.

C.3.2 Handling Input and Output - GameIO

We will now show how one of the above interactors, the *GameIO* process, can be defined. The *GameIO* process shown below performs two purposes. It receives input from the user and passes it to *App*. It also receives pictures from *App* and displays them to the user. This is, therefore, a simplified version of a CNUCE interactor. The Collection and Presentation have been combined into one component GameOutput. The Measure and Abstraction remain in modified form as *GameInput* and *Buffer*. As there is no immediate feedback from user input, the *GameInput* and *GameOutput* processes are unconnected. Output sent to the interactor should be immediately displayed, so we assume that an output collect action is also an output trigger.

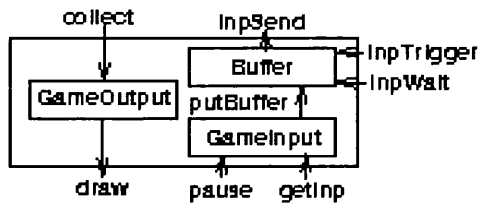

Figure 55 - The GameIO Interactor

The *GameInput* process can be specified as follows:

```
process GameInput [putBuffer,getInp,pause]
            (kb:Keyboard):noexit :=
   (pause;putBuffer!Pause;
    pause;putBuffer!Unpause;
    GameInput [putBuffer,getInp,pause] (kb))
  []
   (getInp!kb?inp:KeyboardEv;
    ([valid inp] -> putBuffer!(interpret inp);
            GameInput [putBuffer,getInp,pause] (kb)
     []
     [not (valid inp)] ->
            GameInput [putBuffer,getInp,pause] (kb)))
 endproc
```

It either collects input events and places valid ones in a buffer, or is blocked by a pause event until another pause event occurs. Pause and unpause events, are also placed as input into the buffer. This LOTOS code is fairly expressive. However, In some places it uses an unusual syntax. Conditional expressions (such as [valid inp] -> ... [] [not (valid inp)[-> ...) are expressed in a way that will be unfamiliar to most programmers. This may make the language less visually appealing.

The *Buffer* process can be specified as follows:

```
process Buffer [putBuffer,inpTrigger,inpWait,inpSend]
            (ns:seq InputEvent):noexit:=
   inpTrigger;inpSend!(first ns);Buffer [...] (rest ns)
 [] putBuffer?inp:Input;Buffer [...] (add inp ns)
 [] inpWait;
     [null ns] -> putBuffer?inp:Input;
                  inpSend!inp;
                  Buffer [...] ns
   [] [not (null ns)] -> inpSend!(first ns);
                         Buffer [...] (rest ns)
endProc
```

It allows the application (*App*) to acquire input in one of two ways. The *inpTrigger* event may fire. This means that the application asks for the next piece of input. If there is none, it merely gets back the value *Nothing*, otherwise it gets back the first input event. Alternatively, the application may use *inpWait*. This causes it to wait until some piece of input is actually available. The *inpTrigger* and *inpWait* actions are both used by *App*. The Game should continue whether the user sends input or not, therefore usually App will use *inpTrigger*. However, when *App* receives a pause event, it should block until an unpause event. This is done, by using *inpWait*.

The Buffer process is an example of asynchronous communication in LOTOS. If two processes communicate via it, the sender need not wait for the receiver before it can continue. We shall exploit this fact in the conversion to Haggis code.

The *GameOutput* process is much simpler, it collects pictures from the application and draws them on the screen.

```
process GameOutput[collect,draw] (screen:Screen):noexit :=
    collect?pic:Picture;
    draw!pic;
    GameOutput[collect,draw] screen
endproc
```

The complete interactor is therefore formed, by running in parallel the three components:

```
process GameIO[collect,draw,pause,getInp,inpSend,inpTrigger,inpWait] :=
  hide putBuffer in
   (GameInput[putBuffer,getInp,pause] kb
       |[putBuffer]|
    Buffer[putBuffer,inpTrigger,inpWait,inpSend] (emptyBuffer))
   |||
   GameOutput[collect,draw] screen
endproc
```

From this specification, we can clearly see, and reason about, how the *GameIO* process operates, and interacts with other processes.

C.3.3 The Rest of the System

The *Pause*, *Restart* and *Quit* interactors are simply instances of the button interactor defined earlier. The Score interactor is simply a label that behaves identically to *GameOutput*.

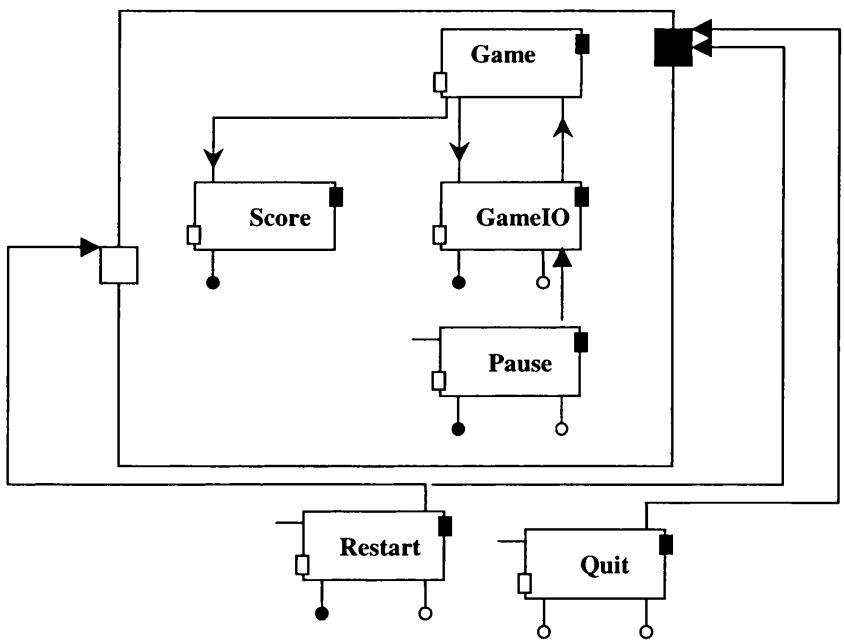

Figure 56 - The Interactor Network for the Game

The interactors can be represented as an interactor network shown in Figure 56. This demonstrates how the interactors can be combined together, by linking their inputs and outputs. Black circles represent an interactor being connected to the user output, white circles to user input. This provides another way of understanding how the system fits together.

We therefore have a clear, high level specification which designers may use to reason about a proposed system. They could, for instance, use a tool such as Caesar/Aldebaran [67], to prove that their specification preserved certain predefined properties. Design errors could therefore be found prior to implementation.

There are further benefits to the use of interactors. The modularity of the specification makes it easy to replace individual agents. For instance, we could easily reuse the interface components with a different application. To do this we could replace the application process (*App*), and then with only minimal modification, of the *GameIO* interactor (to respond to any change of input keys), we could produce an entirely different game.

C.3.4 Specifying The Data Types And Operations

In common with Alexander [6], we use an executable subset of VDM to define the data and operations in our specification. This allows a more easily programmable notion of state with invariants and operations. This approach proves particularly useful when specifying the application (*App*) process, which maintains data about the lasers, enemies, ship and background. We have, for instance, access to sequences to maintain the data about collections of lasers and enemies.

As an example, we will show how the ship data can be specified. Its position, width and height are maintained using a rectangle data type. The ship also has an image *pic,* which will be displayed at its current position, using the *shipPic* function. There is an invariant (*inv*). The ship must remain on screen at all times, that is its *x* position must be greater than 0 and less than the width of the screen (minus its own width, so that all of the ship is displayed). The equivalent is true for its *y* position with respect to screen height and its own height. The ship's data is all packaged up within a Haskell record. The invariant will be applied to any attempt to alter the record, for instance, when we try to move the ship. This would appear in Haskell as follows:

```
data Ship = Ship {rect :: Rectangle,pic :: Picture}
inv ship =
 let (Rect x y w h) = rect ship
 in
 x >= 0 && x <= screenWidth - w && y < screenHeight - h && y >= 0

shipPic ship = let (Rect x y w h) = rect ship in placeAt (x,y) pic
```

In Haskell record fields, such as *rect*, are applied to a record as functions, using the syntax *rect* ship rather than with the more common *ship.rect* syntax.

We therefore abandon the ACT ONE algebraic data specification language, normally used with LOTOS, as it lacks modularity. The new LOTOS standard, E-LOTOS, itself replaces ACT ONE with a more functional style because of the difficulty programmers had with it [Jeffrey1996]. Our approach overcomes some of the problems with LOTOS interactors, as they now have both a behaviour and internal state, but with the behaviour still clearly visible on top.

We compose the components (lasers, enemies, ship and background) together in a functional way. However, the solution is not perfect. Each component should change over time. However, there is a single control component, the *App* process, which accepts all user input and passes it to the collection of components. We would have a more modular solution, if we could make each individual component dynamic, and then compose them together.

C.4 Conversion to Haggis

We can convert LOTOS specifications into Haggis code as follows:

- events become Haggis I/O actions; these IO actions need to be defined. They can be presentation layer events, defined using structured graphics, and Haggis interaction objects. They can perform communication between interactors, or can alter the system state (application layer);
- LOTOS communication is converted into Haskell code using our extended concurrency library. This provides programmers with synchronous communication, through a library of LOTOS like operators. We also include asynchronous communication along channels. This provides a more efficient and, arguably, a more elegant way of implementing any asynchronous communication in a LOTOS specification. Asynchronous and synchronous communication can be combined freely within this system. Synchronous LOTOS communication becomes synchronous Haskell

communication, and asynchronous communication, simulated in LOTOS, becomes asynchronous Haskell communication;

- the data manipulation operations are already executable.

Haggis provides a number of abstractions that make this conversion easier. This is important if specifications and prototypes are to be cost effective. We make use of the button abstraction, to allow an implementation of the button interactors. This abstraction is equivalent to the interactor described earlier. It takes a picture to display (*pic*), and a value (*n*) to return on any successful button click; it provides *setLabel*, and *getButtonClick* operations, and provides equivalent feedback on user actions.
The Pause interactor can be seen below. Input is sent from the interactor via the *pauseBtn* channel. Another process can receive this data by using the *getButtonClick pauseBtn* operation. The trigger that causes input to be sent, is a button click.

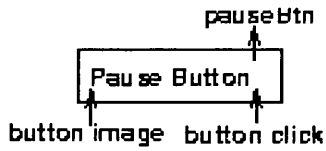

Figure 57 - Haggis Pause Button

We can translate the *GameIO* process fairly simply. The *Buffer* process can be replaced by an asynchronous channel, as it is an example of a LOTOS specification, of asynchronous communication. Specifically, consider a channel with three operations *sendChannel*, which sends a value along the channel, *waitChannel*, which blocks until a value is received, and *getChannel*, which either returns the next value from the channel, or *Nothing* if the channel is empty. Placing values into the buffer (*putBuffer*) can be done with *sendChannel*. The Buffer will be triggered by *waitChannel* or *getChannel*, either of which will cause input to be sent out from the buffer. *waitChannel* is equivalent to the *inpWait* branch of the *Buffer* process; *getChannel* to the *inpTrigger* part of the *Buffer* process.

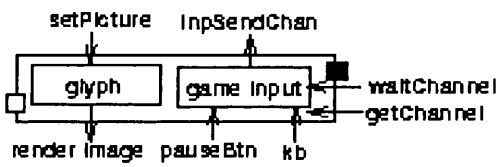

Figure 58 - Haggis Game IO Interactor

The *GameIO* interactor would therefore function as shown above. The *GameInput* process sends input via the *inpSendChan* channel. Input is taken from this channel by either a *waitChannel* or a *getChannel* operation. These operations use the *inpSendChan*. Input is received from both the Pause button (via the *pauseBtn* channel) and the Keyboard (via the *kb* channel). The *GameInput* process can therefore be defined in Haggis as follows:

```
gameInput :: Button ()
          -> Keyboard
          -> Channel InputEvent
          -> EventIO ()
gameInput pauseBtn kb inpSendChan =
 choose
  (event (getButtonClick pauseBtn) -=> do
    sendChannel inpSendChan Pause
    receive (getButtonClick pauseBtn)
    sendChannel inpSendChan Unpause
    gameInput pauseBtn kb inpSendChan)
  .|.
  (event (getKeyboardEv kb) ==> \inp ->
    if valid inp then do
     sendChannel chan (interpret inp)
     gameInput pauseBtn inpSendChan kb
    else
     gameInput pauseBtn inpSendChan kb)
```

The .|. symbol is our choice operator. The expression *event e* ==> \x -> *a*, means perform event *e*, and then let *x* equal the result of *e* in action *a*. Choice can take place between event guarded expressions. The syntax is similar to that used in Functional Reactive Programming.

We can implement the *GameOutput* process very simply as a Glyph. It receives pictures to display via *setPicture*, and renders them on the screen.

The above translation has shown that we can easily convert the interactors specification into executable code. We have therefore dealt with the behaviour of the system. As a final step, we will show how to describe what the system looks like, and so build the full screen shown earlier. Again, we are concerned with supporting full graphical interaction as easily as possible.

We build the screen with six types of operation. The *mkDC* operation creates a Display Context *dc*, which contains information about style values and the window that will be created. The *button* operation creates a button as described earlier. The *label* operation creates a label displaying the given string, which may be updated (using the *lbl* handle). The *glyph* operation creates a simple output area containing the specified picture. The *catchKeyboardEv* operation makes the *glyph* interactive and able to receive keyboard events (via the *kb* channel). Finally the *realiseDH* operation renders the components on to a window using the *vbox* combinator to place *Display Handles* above one another, and using *hbox* to place *Display Handles* next to one other.

```
screen = do
           dc <- mkDC [];
           (rbutton,rdh) <- button (text "Restart") True dc;
           (pbutton,pdh) <- button (text "Pause") True dc;
           (qbutton,qdh) <- button (text "Quit") True dc;
           (_,ldh) <- label "Score:" dc;
           (lbl,ldh2) <- label "0" dc;
           (gl,screendh) <- glyph screenImage dc;
           (kb,sdh) <- catchKeyboardEv screendh
           realiseDH (hbox [sdh,vbox[ldh,ldh2,pdh,rdh,qdh]])
```

The separation provided here between the appearance of the interface, and the underlying behaviour is a powerful feature. It adds to the separation between the interface and application. However, it is only fully possible as we have a fairly static interface.

The system can now be run, and tested on users. At this stage, problems with the interface can be discovered and used to reshape the initial specification. For example, early prototypes of the system did not provide a pause button. This was clearly necessary to allow users to abandon the game temporarily.

C.5 Conclusions

Through our short example, we have shown that functional languages make good tools for transforming high-level concurrent specifications into executable code. In particular, the Haggis system provides a high level compositional concurrent interface that makes this fairly. We start with a high level, modular, LOTOS specification that can be used to reason about possible interaction problems in a system, using tools such as LITE. We can then easily transform this into executable Haggis code. The resulting executable system supports user testing. The high level specification can be used to target resulting evaluations. This all provides for an iterative, user centred approach to design.

However, the LOTOS-Haggis combination is not perfect. The specification is purely event based. While each interactor can have a fairly complex internal state, sharing this state between interactors is more difficult. Haggis does not permit *Declarative Concurrency*, by permitting constraints to be defined between evolving processes. It does not support dynamically evolving structured graphics. This makes it more difficult to build systems in a structured way. Some aspects of the LOTOS specification may not be very visually appealing. Because of this purely event based approach, it can be difficult to consider some usability properties, such as visibility. This is because it is difficult to understand the mapping from the state of components to the interface. Finally, there are still some problems in providing a smooth combination of notation and implementation as all predefined Haggis components communicate in an asynchronous, rather than a synchronous manner.

Glossary[33]

Algebraic type An algebraic type definition states what are the **constructors** of the type. For instance, the declaration

```
data Tree a = Leaf Int
            | Node Tree Tree
```

says that the two constructors of the Tree type are Leaf and Node, and that their types are, respectively,

```
Leaf :: Int -> Tree,
Node :: Tree -> Tree -> Tree.
```

Application This means giving values to some or all arguments of a function. If an n-argument function is given fewer than n arguments this is called **partial application**.

Action An IO action in Haskell is a value of type `IO a`, which performs some side-effecting operation (such as printing to or reading from a file) and returns some value, `a`.

Behavior In Functional Reactive Programming a Behavior is a time-varying value. The simplest conceptual model is:

```
type Behavior a = Time -> a
```

Callback An action passed to an object that can be performed by the recipient. For instance, a button is passed a callback action which it performs when it is clicked.

Combinator Another name for a function.

Constraint A definition that states that one value depends on another. For instance, we could have a graphical constraint that defined a box that surrounded a circle. With a constraint style of programming, when a value is changed, all those values that depend on it should be updated.

Constructor An **algebraic type** is specified by its constructors, which are the functions which build elements of the algebraic type.

Context The definition which appears before `=>` in type and class declarations. A context `M a`, means that the type `a` must belong to the class `M` for the function or class definition to apply. For instance, `Eq a` means that a value must have equality defined upon it.

Curried function A function of at least two arguments, which takes its arguments one at a time, rather than in a tuple.

Derived class instance An instance of a standard class which is derived by the compiler rather than by the programmer.

Event An event represents a stream of values that occur at discrete points in time. An event can be used to model concepts such as mouse clicks. Every time the mouse is pressed, the event stream will generate an **occurrence.**

Functional Reactive Programming A declarative programming style based around the use of **behaviors** and **events**.

Higher-order function A function is higher-order if any of its arguments, or its result are themselves functions.

Imperative concurrency A functional GUI that uses an approach based on imperative concurrency, structures user interface components as a set of processes, that execute concurrently and consume user input. Here processes are created, and their behavior is defined, using IO actions. Haggis is a good example of such a system.

Lambda expression An expression which denotes a function. After a '\' we list the arguments of the function, then an '->' and then the result. For instance, the following lambda expression adds two numbers together.

```
\x y -> x + y
```

Lazy evaluation In a function application only those values which are needed will be evaluated, and only the parts of data structures which are needed will be examined.

Memoization Keeping the value of a sub-computation (in a map for instance) so that it can be reused rather than recomputed.

[33] The descriptions for a number of these terms are taken from [201].

Monad A monad consists of a type class with at least two functions, `return` and `>>=`. Informally, a monad can be seen as performing some form of action before returning a result. The first monad function simply returns a given value; the second sequences two monadic operations.

Occurrence An event occurrence is a (Time,value) pair that is generated by an event when it occurs. For instance, a mouse motion event generates occurrences with a time and a mouse location, whenever the mouse is moved.

Partial application If an `n`-argument function is given fewer than `n` arguments this is called partial application. The application is partial, because the result can itself be passed further parameters.

Pure programming language A functional programming language is pure if it does not allow side-effects. For a given argument, a pure function returns the same value irrespective of when it is called.

Semantic wiring Within user-interface code the semantic wiring connects user input from a widget to the application code. In contrast, geometric composition involves organising the layout of a screen. FranTk separates these two concepts.

Side effect If evaluating an expression can cause other things to happen, besides a value being produced, then that function is impure. In Haskell such side-effecting actions can only happen within a **monad**.

Stream A stream is a channel upon which items arrive in sequence. In Haskell we can think of lazy lists in this way.

Stream processing A functional GUI system that uses stream processing, views user interface components as stream processors, that consume streams of user input and produce streams of output commands. Fudgets and Gadgets are two systems that take this approach.

Type class A collection of types. A class is defined by specifying a signature; a type is made an instance of the class by supplying an implementation of the definitions of the signature for the type.

References

[1] Abowd, G. *Agents: Communicating Interactive Processes.* In D. Diaper et al (eds.) *Human Computer Interaction - INTERACT '90*, 1990, North Holland Press.

[2] Abowd, G. and A. Dix *Giving undo attention.* Interacting with Computers, 1992. **4**(3):317-342.

[3] Aitken, J.S., et al., *Interactive Theorem Proving: An Empirical Study of User Activity.* Journal of Symbolic Computation, 1998. **25**(2): p. 263-284

[4] Alexander, H. *Executable specifications as an aid to dialogue design.* in *Human Computer Interaction - INTERACT'87.* 1987. Stuttgart University, Germany: North Holland.

[5] Alexander, H. (1990), *Structuring dialogues using CSP*, in *Formal methods in Human computer interaction*, M. Harrison and H. Thimbleby, Editors. 1990, Cambridge University Press

[6] Alexander, H. and V. Jones, *Software design and prototyping using me too.* 1990: Prentice Hall

[7] Allen, S. *Visual Tcl*, 1996. available from http://vtcl.sourceforge.net/

[8] Andersen, H.R. *Model Checking and Boolean Graphs.* Theoretical Computer Science, 1994. **126**(1): p. 3-30.

[9] Apple Computer Inc., *Hypercard Reference.* 1990: Claris Corportation.

[10] Austin, P., *DTI Report on Formal Methods in Industry*, 1993: UK Department of Trade and Industry.

[11] Baecker, R. *et al. The User-Centred Iterative Design of Collaborative Writing Software.* in Proceedings of *INTERCHI'93.* 1993. p. 399-405. Amsterdam, Holland: IOS Press and ACM Press.

[12] Bastide, R. *et al. Integrating Rendering Specifications into a Formalism for The Design of Interactive Systems.* in *Design, Specification and Verification of Interactive Systems'98.* 1998. p.171-191. Abingdon, UK: SpringerWienNewYork.

[13] Bentley, R. *Supporting multi-user interface development for cooperative systems*, PhD thesis, 1994, Lancaster University.

[14] Bevan, N. and M. Macleod, *Usability Measurement in Context.* Behaviour and Information Technology, 1994. **13**(1,2): p. 132-145.

[15] Bird, R. and P. Wadler, *Introduction to Functional Programming*, Prentice Hall International Series in Computer Science, ed. C.A.R. Hoare. 1988: Prentice Hall.

[16] Bowers, J. *The Work to Make a Network Work: Studying CSCW in Action. Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work.* 1994. p. 287-298. Chapel Hill, United States: ACM Press.

[17] Bowers, J. and J. Pycock, 1994, *Talking through design: requirements and resistance in cooperative prototyping.* in *Proceedings of CHI'94.* 1994. p. 299-305. Boston, United States: ACM Press.

[18] Breedvelt-Schouten, I.M., F.D. Paterno, and C.A. Severijns. *Reusable structures in task models.* in *Design, Specification and Verification of Interactive Systems'97.* 1997. p. 225-241. Granada, Spain: SpringerWienNewYork.

[19] Brown, J., T.C.N. Graham, and T. Wright. *The Vista environment for the coevolutionary design of user interfaces.* in *Proceedings of CHI'98.* 1998. p. 376-383. Los Angeles, USA: ACM Press.

[20] Button, G. and P. Dourish. *Technomethodology: paradoxes and possibilities.* in *Proceedings of CHI'96.* 1996. p. 19-26. Vancouver, Canada: ACM Press.

[21] Campos, J.C. and M.D. Harrison, *Formally verifying interactive systems: A review.* in *Design, Specification and Verification of Interactive Systems'97.* 1997. p. 109-125. Granada, Spain: SpringerWienNewYork.

[22] Campos, J.C. and M.D. Harrison, *The Role of Verification in Interactive System Design.* in *Design, Specification and Verification of Interactive Systems'98.* 1998. p. 155-171. Abingdon, UK: SpringerWienNewYork.

[23] Campos, J.C. and M.D. Harrison *Using Automated Reasoing in the design of an audio-visual communication system.* In *Design, Specification and Verification of Interactive Systems'99.* 1999. p. 167-188. : SpringerWienNewYork.

[24] Campos, J.C. and M.D. Harrison (1999), *From Interactors to SMV: A Case Study in the Automated Analysis of Interactive Systems*, 1999, York University Technical Report, ftp://ftp.cs.york.ac.uk/reports/YCS-99-317.ps.gz

[25] Carlsson, M. and T. Hallgren: *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*, PhD thesis, 1998, Chalmers University of Technology.

[26] Chehaibar, G. *et al. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS.* in *Proceedings of. the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96.* 1996. p. 435-450. Kaiserslautern, Germany.

[27] Claessen, K., T. Vullinghs and E. Meijer. *Structuring Graphical Paradigms in TkGofer.* in *Proceedings of. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97).* 1997. p. 251-262. : ACM

[28] Clark, H. and S. Brennan, *Grounding in Communication*, in *Readings in Groupware and Computer-Supported Cooperative Work, assisting human-human collaboration,* R. Baecker, Editor. 1993, Morgan Kaufman Publishers.

[29] Clark, S. *Literate Development*, PhD thesis, 1997, University of Glasgow.

[30] Conklin, J. and M.L. Begeman *gIBIS: A Tool for All Reasons.* Journal of American Society for Information Science, 1989. **40**(3): p. 200-213.

[31] Coplien J. and D. Schmidt*: Pattern Languages of Program Design*, 1995, Addison-Wesley.

[32] Joelle Coutaz (1997) PAC-ing the Architecture of Your User Interface, in Proceedings of the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems, Springer Verlag, 1997.

[33] Courtney, A. Frappe: FRP in Java, draft submitted to ICFP'2000.

[34] Cuomo, D *Understanding the applicability of sequential data analysis techniques for analysing usability data.* Behaviour & information technology, 1995. **13**(1-2): p. 171-182.

[35] Daniels, A. *A semantics for functions and behaviors*, PhD Thesis, 1999, University of Nottingham.

[36] Desurvire, H.W. J.M. Kondziela and M.E. Atwood, *What is Gained and Lost when Using Evaluation Methods Other than Empirical Testing.* in *Proceedings of the HCI'92 Conference on People and Computers VII.* 1992. p. 89-102.

[37] Dix, A. *Formal Methods for Interactive Systems*, 1991, Academic Press.

[38] Dix, A. and G. Abowd, (1996) *Modelling status and event behavior of interactive systems.* Software Engineering Journal, 1996. **11**(6): p. 334-346.

[39] Dourish, P. and V. Bellotti *Awareness and Coordination in Shared Workspaces.* in *Proceedings of ACM CSCW'92 Conference on Computer-Supported Cooperative Work.* 1992. p. 107-114. : ACM Press

[40] Draper, S.W. *The notion of task in HCI.* in *Proceedings of. INTERCHI'93.* 1993. p. 207-208. Amsterdam, Holland: IOS Press and ACM Press.

[41] David Duke(1995), Michael Harrison, Joelle Coutaz, Laurence Nigay, Daniel Salber, Giorgio Faconti, Menica Mezzanotte, Fabio Paterno and David Duce, *Theoretical Framework with Reference Model and Multi-Agent Presentations*, ESPRIT Basic Research Action 7040 Amodeus Project deliverable, document D9.

[42] Dutt A., H. Johnson and P. Johnson, *Evaluating Evaluation Methods.* in *Proceedings of the HCI'94 Conference on People and Computers IX.* 1994: Cambridge University Press.

[43] Eijk, P. v. and H. Eertink. *Design of the LotosPhere symbolic LOTOS simulator.* in *Proceedings of Formal Description Techniques, III - Proceedings of the FORTE 90 Conference.* 1991 Amsterdam: North-Holland.

[44] Elliott, C. and P. Hudak. *Functional Reactive Animation.* in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming: ICFP'97.* 1997. p. 263-273. Amsterdam: ACM.

[45] Elliott, C. (1997b), *Modeling Interactive 3D and Multimedia Animation with an Embedded Language.* in *Proceedings of. Proceedings of the Conference on Domain-Specific Languages (DSL-97).* 1997. p. 285-296. : USENIX Association.

[46] Elliott, C. *Functional Implementations of Continuos Modeled Animation.* in *Proceedings of. PLILP/ALP 1998.* 1998. p. 284-299. Lecture Notes in Computer Science, 1490, : Springer Verlag

[47] Erkok, L. and J. Launchbury. *Recursive monadic bindings.* in *Proceedings of. Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming: ICFP'00.* 2000. p174-185. Montreal, Canada: ACM Press.

[48] EUROCONTROL, *EATCHIP Phase III HMI Catalogue.* Available at http://www.eurocontrol.be/projects/eatchip/hmi/, 1998.

[49] EUROCONTROL, *ATM Strategy for 2000+,* Volume 2. Available at http://www.eurocontrol.be/projects/eatchip/atmstrat/, 1998.

[50] Faconti, G.P. and e. al, *Graphical Process Interaction Networks for Lotos Parallel Expressions,* . 1993, Amodeus Project Document, SM/WP25.

[51] Fath, J., T. Mann, and T. Holzman, *A practical guide to software usability labs: lessons learned at IBM.* Behaviour & information technology, 1994. **13**(1-2): p. 94-105.

[52] Fernandez, J.-C., Kerbrat, and L. Mounier. *Symbolic Equivalence Checking.* in *Proceedings of the 5th Workshop on Computer-Aided Verification.* 1993 Lecture Notes in Computer Science, 697, Heraklion, Greece: Springer-Verlag.

[53] Fernandez J.-C. and L. Mounier. *A Tool Set for Deciding Behavioral Equivalences.* in *Proceedings of. Proceedings of CONCUR'91.* 1991 Amsterdam, Netherlands.

[54] Fernandez , J.-C. and L. Mounier, *A Local Checking Algorithm for Boolean Equation Systems,* . 1995, Rapport SPECTRE, 95-07, VERIMAG, Grenoble.

[55] Fields, R.E. (P.C. Wright, and M.D. Harrison, *A framework for refining the environment of an interactive system to an artefact,* 1994. Unpublished, available at http://dcpu1.cs.york.ac.uk:6666/~bob/papers.html

[56] Fields, R.E. N. Merriam, and A. Dearden. *DMVIS: Design, modelling and validation of interactive systems.* in *Design, Specification and Verification of Interactive Systems'97.* 1997. p. 29-45. Granada, Spain: SpringerWienNewYork.

[57] Fields, R.E. and N. Merriam. *Modelling in Action. Reports from the DSVIS'97 working groups.* in *Design, Specification and Verification of Interactive Systems'97.* 1997. p. 307-320. Granada, Spain: SpringerWienNewYork.

[58] Fields, R.E. N Merriam (1998), *Inference and Information Resources: A Design Case Study.* in *Proceedings of. Design, Specification and Verification of Interactive Systems'98.* 1998. P41-57. Abingdon, UK: SpringerWienNewYork.

[59] Finne, S., and S.L. Peyton Jones. *Pictures: A simple structured graphics model.* in *Proceedings of Glasgow Functional Programming Workshop.* 1995 Ullapool.

[60] Finne, S. and S.L. Peyton Jones. *Composing the User Interface with HAGGIS.* in *Proceedings of. Summer School on Advanced Functional Programming.* 1996. p. 1-37. Lecture Notes in Computer Science, 1129, Olympia, WA..

[61] Finne, S. and S.L. Peyton Jones: Composing the User Interface with HAGGIS, *Summer School on Advanced Functional Programming*, Olympia, WA, Aug 25-30, Springer Verlag LNCS, 1996.

[62] Finne, S. *Composing Graphical User Interfaces in a Purely Functional Language*, PhD thesis, 1998, University of Glasgow.

[63] Fowler, C., *et al., Using the usability laboratory: BT's experiences.* Behavior and Information Technology, 1994. **13**(1,2): p. 146-153.

[64] Fuchs N.E., *Specifications are (preferably) executable.* Software Engineering Journal, 1992. **7**(5): p. 323-334.

[65] Garavel H. and R.-P. Hautbois. *An Experiment with the LOTOS Formal Description Technique on the Flight Warning Computer of Airbus 330/340 Aircrafts*. in *Proceedings of First AMAST International Workshop on Real-Time Systems*. 1993 Iowa City, Iowa, USA

[66] Garavel, H. *An Overview of the Eucalyptus Toolbox*. in *Proceedings of the COST 247 Interanational Workshop on Applied Formal Methods in System Design*. June 1995. p. 76-88. University of Maribor, Slovenia.

[67] Garavel, H., *et al. CADP'97 – Status, Applications and Perspectives*, in *Proceedings of 2nd COST 247 International Workshop on Applied Formal Methods in System Design*. June 1997, Zagreb, Croatia.

[68] Gill, A. *Debugging Haskell by Observing Intermediate Data Structures*. in *Proceedings of. Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*. September 2000. p. 70-82. Montreal.

[69] Goldson, D. *Abstract modelling of interactive Systems. Human-Computer Interaction – INTERACT'97*, July 1997, p.134-142, Sydney, Australia: Chapman & Hall.

[70] Gordon , A.D. and K. Hammond. *Monadic I/O in Haskell 1.3*. in *Proceedings of 1995 ACM SIGPLAN Haskell Workshop*. 1995. p. 50-69. La Jolla, California..

[71] Graham, T.C.N., *Declarative Development of Interactive Systems*. Berichte der GMD. Vol. 243. 1995, Munich: R. Oldenbourg Verlag.

[72] Graham, T.C.N. and T. Urnes. *Linguistic Support for the Evolutionary Design of Software Architectures*. in *Proceedings of. Eighteenth International Conference on Software Engineering*. 1996. p. 418-427. Berlin, Germany: IEEE Computer Society Press.

[73] Graham, T.C.N., *et al.*, *The Clock Methodology: Bridging the Gap Between User Interface Design and Implementation*, York University Technical Report CS-96-04. York University, August 1996.

[74] Graham, T.C. T.C.N. and T. Urnes. *Semi-replicated implementations of a distributed architecture*. in *Proceedings of. Design, Specification and Verification of Interactive Systems'99*. 1999: SpringerWienNewYork.

[75] Gray, P. D. England, and S. McGowan, *XUAN: Enhancing the UAN to capture Temporal Relations among Actions*, Technical Report IS-94-02, Department of Computing Science, University of Glasgow.

[76] Green T. *Cognitive dimensions of notations*. in *Proceedings of. People and Computers IV*. 1989. p.443-460 : Cambridge University Press.

[77] Greenberg , S. and D. Marwood. *Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface*. in *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*. 1994. p. 207-217. : ACM Press.

[78] Griffiths, T. and e. al. *An Open-Model-Based Interface Development System: The Teallach Approach*. in *Draft Proceedings of. Design Specification and Verification of Interactive Systems'98*. 1998 Abingdon, UK.

[79] Grudin, J. *Why CSCW applications fail: problems in the design and evaluation of organisational interfaces*. in *Proceedings of ACM CSCW'88 Conference on Computer-Supported Cooperative Work*. 1988. p. 85-93. Portland, Oregon: ACM Press.

[80] Grudin J., *Obstacles to user involvement in software product development, with implications for CSCW*. International Journal of Man Machine Studies, 1991. **34**(3): p. 435-452.

[81] Grudin, J., *Groupware and social dynamics: Eight challenges for developers*. Communications of the ACM, 1994. **37**(1): p. 92-105.

[82] Hall, A. *Do interactive systems need specifications*. in *Design, Specification and Verification of Interactive Systems'97*. 1997. p. 1-13. Granada, Spain: SpringerWienNewYork.

[83] Halverson, C. *Distributed Cognition as a Theoretical Framework for HCI: Don't Throw the Baby out with the bathwater, the importance of the cursor in Air Traffic Control*, technical report 9403, Department of Cognitive Science, University of California.

[84] Harper, R., J. Hughes, and D. Shapiro, *Harmonious working and CSCW: computer technology and airtraffic control*, in *Studies in CSCW: Theory, Practice and Design*, J.Bowers and S. Benford, Editors. 1991, North Holland: Amsterdam.

[85] Harrison , M.D. and D.J. Duke. *A review of formalisms for describing interactive behaviour.* in *Proceedings of. ICSE'94 Workshop on SE-HCI.* 1994 Lecture Notes in Computer Science, 896: Springer Verlag.

[86] Hartson, H.R., A.C. Siochi, and D. Hix, *The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs.* ACM Transactions on Information Systems, 1990. **8**(3): p. 191-203..

[87] Hayes, I.J. and C.B. Jones, *Specifications are not (necessarily) executable.* Software Engineering Journal, 1989. **4**(6): p. 330-338

[88] Heimdahl, , M. and N. Leveson, *Completeness and Consistency in Hierarchical State-Based Requirements.* Transactions on Software Engineering, 1996. **22**(6): p. 363-377.

[89] Henderson, R., *et al., An examination of four user based software evaluation methods.* Interacting with computers, 1995. **7**(4): p. 412-432.

[90] Hill. R. *The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications.* in *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems.* 1992. p.335-342. : ACM Press.

[91] Hill, R. *The RENDEZVOUS Constraint Maintenance System.* in *Proceedings of. ACM Symposium on User Interface Software and Technology.* 1992. p.225-234. : ACM Press.

[92] Hoc, J.M., T.R.G. Green, R. Samurcay and D.J. Gilmore (eds), *Psychology of Programming,* Computers and People Series, 1990, Academic Press Ltd.

[93] Hoiem, D. and K. Sullivan, *Designing and using integrated data collection and analysis tools: challenges and considerations.* Behavior and information technology, 1994. **13**(1,2): p. 160-170.

[94] Hudak, P. *The Haskell School of Expression, Learning Functional Programming Through Multimedia,* 2000, Cambridge University Press.

[95] Hughes, , J., *et al. Moving out from the Control Room: Ethnography in System Design.* in *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work.* 1994. p. 429-439. ACM Press.

[96] Javaux, D. and P Polson, *A method for predicting errors when interacting with Finite State Machines,* to appear in Elsevier Reliability Engineering and System Safety Journal.

[97] Jeffrey, A. and G. Leduc, *E-LOTOS core language.* Output of the Kansas City meeting, version 1996/09/20, (ISO-IEC/JTC1/SC21/WG7).

[98] John, B. and H. Packer. *Learning and Using the Cognitive Walkthrough Method: A Case Study Approach.* in *Proceedings of CHI'95.* 1995. p. 429-436 : ACM Press.

[99] John, B and S.J. Marks *Tracking the Effectiveness of Usability Evaluation Methods,* Computer Science Technical Report CMU-CS-96-160, 1996, Human Computer Interaction Institute, Carnegie Mellon University, USA.

[100] Johnson, C.W. and M.D. Harrison, *Using temporal logic to support specification of interactive control systems.* International Journal of Man-Machine Studies, 1992. **37**(3): p. 357-385.

[101] Johnson, C.W. *Literate specifications,* Software Engineering Journal, 1996, **11**(4): p. 225-237.

[102] Johnson, C.W. *Utility of User interface notations,* submitted to the Journal of Human Computer Interaction, 1997.

[103] Johnson, P., S Wilson, P Markopoulos & J Pycock *ADEPT - Advanced design environment for prototyping with task models,* Demonstration abstract in *Proceedings of INTERCHI'93,* 1993, pp 56. ACM Press.

[104] Jones, M.P. *Type Classes with Functional Dependencies.* in *Proceedings of. 9th European Symposium on Programming: ESOP 2000.* 2000. Lecture Notes in Computer Science, 1782, Berlin,Germany: Springer-Verlag.

[105] Jones S. and J. Sapford, *The Role of Informal Representations in Early Design.* in *Proceedings of. Design, Specification and Verification of Interactive Systems'98.* 1998 p.117-134. Abingdon, UK: SpringerWienNewYork.

[106] Kahl, W., O. Braun, and J. Scheffczyk, *Editor Combinators - A First Account*, Technical Report Nr. 2000-01, 36 pages Fakultät fur Informatik Universität der Bundeswehr Munchen, June 2000.

[107] Karat, C.M., R. Campbell and T. Fiegel 1992, *Comparison of empirical testing and walkthrough methods in user interface evaluation.* in *Proceedings of CHI'92*. 1992. p. 397-404. : ACM Press.

[108] Kerbrat, A. and S.B. Atallah. *Formal Specification of a Framework for Groupware Development,*. in *Proceedings of. Proceedings of the 8th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols FORTE'95.* 1995Montreal, Canada.

[109] Knutson, J., T. Anand and R. Henneman, *Evolution of a User Interface Design: NCR's Management Discovery Tool (MDT).* in *Proceedings of CHI'97*. 1997. p. 526-553 : ACM Press.

[110] Krasner, G. and S. Pope, *A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1988. **1**(3): p. 26-49.

[111] Krimm , J.-P. and L. Mounier. *Compositional State Space Generation from Lotos Programs.* in *Proceedings of. Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems.* 1997University of Twentee, Enschede, The Netherlands.

[112] Kyng , M., *Designing for cooperation: cooperation in design.* Communications of the ACM, 1991. **34**(12): p. 65-73.

[113] Landay, J. and B. Myers. *Interactive Sketching for the Early Stages of User Interface Design.* in *Proceedings of CHI'96*. 1995. p. 43-50: ACM Press.

[114] Lauesen, S. *Usability Engineering in Industrial Practice. Human-Computer Interaction – INTERACT'97*, July 1997, p.15-22, Sydney, Australia: Chapman & Hall.

[115] Laufer K. *Type Classes with Existential Types.* Journal of Functional Programming, 1996. **6**(3): p. 485-517.

[116] Leveson, N., M. Heimdahl, H. Hildreth and J.D. Reese *Requirements Specification for Process Control Systems.* IEEE Transactions on Software Engineering, 1994. **SE-20**(9): p. 684-707.

[117] Leveson, N., L. Pinnell, S. Sandys, S. Koga, J.D. Reese, *Analysing Software Specifications for Mode Confusion Potential.* in *Proceedings of. Proceedings of Workshop on Human Error and System Development.* March 1997, Glasgow.

[118] Leveson, N . and E. Palmer. *Designing automation to reduce operator errors.* in *Proceedings of the IEEE Systems, Man and Cybernetics Conference.* 1997.

[119] Leveson, N. *et al., Safety Analysis of Air Traffic Control Upgrades*, NASA funded project report, September 1997.

[120] Leveson, N. .D. Reese, and M. Heimdahl. *SpecTRM: A CAD System for Digital Automation.* in *Proceedings of. Proceedings of DASC'98, (Digital Avionics System Conference).* 1998, Seattle.

[121] Lewis, C., P. Polson, C. Wharton and J. Rieman, *Testing a Walkthrough methodology for Theory-based design of Walk-up-and-use Interfaces*, in Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems. 1990. P. 235-242. Seattle, USA: ACM Press.

[122] Maclean, A., R. Young, V. Bellotti, and T. Moran, *Questions, Options and Criteria: Elements of Design Space Analysis*, in [133].

[123] Mackay W.E., R. Guindon, M. Mantei, L. Suchman and D. Tatar. *Video: Data for studying human-computer interaction*, panel discussion in *Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems.* 1998. p.133-137: ACM Press.

[124] Mackay, W.E.. *Ethics, Lies and Videotape...*, in in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems.* 1995. P138-145. Denver, CO, USA: ACM Press.

[125] Mackay, W.E., A.-L. Fayard, L. Frobert and L. Medini. *Reinventing the Familiar: Exploring an Augmented Reality Design Space for Air Traffic Control*, in *Proceedings of ACM CHI'98 Conference on Human Factors in Computing Systems.* 1998. P558-565. Los Angeles, USA: ACM Press.

[126] Macleod, M. *An introduction to usability evaluation*, 1992, *Usability Now!* DTI project report.

[127] Markopoulos, P. *A compositional model for the formal specification of user interface software*, PhD Thesis, 1997, QMW College, University of London.

[128] Mateescu, R.. and Garavel, H. *XTL: A Meta-Language and Tool for Temporal Logic Model-Checking*, in *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98*. July 1998. Aalborg, Denmark.

[129] Microsoft Corporation, Visual Basic Users Guide.

[130] Mitchell A., Posner I. and Baecker I. *Learning to write together using groupware*, in in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*. 1995. P288-295 Denver, CO, USA: ACM Press.

[131] Molich, R. *Preventing user interface distasters*. Behavior and Information Technology, **13:**(1, 2):154-159.

[132] Monk, A., J. McCarthy, L. Watts and O.D. Jones. *Measures of Process*, in *CSCW Requirements and Evaluation*, P.J. Thomas, Editor. 1996, Springer.

[133] Moran, T.P., J.M. Carroll (eds) *Design Rationale: Concepts, techniques and use*, 1996, Hillsdale, Lawrence Erlbaum Assoociates.

[134] Morton, C.A. *Tool Support for Component Based Programming*. York University Technical Report CS-94-02. M.Sc. Thesis, York University, May 1994.

[135] Morris, M., T. Plant, P. Hughes, *CoOp Lab: Practical Experiences with Evaluating a Multi-User System*, in *People and Computers XII: Proceedings of HCI'92*, 1992. p355-368: Cambridge University Press.

[136] Myers, B.A., *et al.*, *Garnet: Comprehensive Support for Graphical Highly-Interactive User Interfaces*. IEEE Computer, 1990. **23**(11): p. 71-85..

[137] Myers, B.A. *Separating application code from toolkits: Eliminating the spaghetti of callbacks.* in *Proceedings of the ACM SIGCHI'91 Conference on User Interface Software Technology*. November 1991. p.211-220 :ACM Press.

[138] Myers, B.A. and M.B. Rosson. *Survey on user interface programming.* in *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, May 1992. p. 195-202: ACM Press.

[139] Myers, B.A., S.E. Hudson and R. Pausch, *Past, Present and Future of User Interface Software Tools*. ACM Transactions on Computer-Human Interaction, 2000, **7**(1):3-28.

[140] Nejabi, R. *Linguistic Support for Developing Groupware Systems*. M.Sc. Thesis, July 1995, Department of Computer Science, York University, Canada.

[141] Newman, W. and M. Lamming, *Interactive System Design*, 1995, Addison Wesley.

[142] Nielsen, J. and R. Molich *Heuristic Evaluation of User Interfaces*, in *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, 1990. p.249-256. Seattle, WA, USA: ACM Press.

[143] Nielsen, J. *Usability Engineering*, 1993, Boston, MA: Academic Press.

[144] Noble, R. *Lazy Functional Components for Graphical User Interfaces*, PhD Thesis, 1995, Department of Computing Science, University of York.

[145] Okasaki, C. *An Overview of Edison*, in *Proceedings of 2000 ACM SIGPLAN Haskell Workshop*, September 2000, p.34-46. Montreal Canada.

[146] Olsen, D.R *Presentational syntactic and semantic components of interactive dialogue specification.* in *User Interface Managements Systems: proceedings of the Workshop on User Interface Systems held in Seeheim.* G.E Pfaff, Editor, p. 125-133. Springer, Berlin, 1985.

[147] Ousterhout, J. *Tcl and the Tk Toolkit*, 1992, Addison-Wesley.

[148] Palanque, P., F. Paterno, R. Bastide, M. Mezzanotte. *Towards an integrated proposal for interactive systems design based on TLIM and MICO*, in *Design, Specification and Verification of Interactive Systems'96*. June 1996. p.162-187, Namur, Belgium: SpringerWienNewYork.

[149] Palanque, P., R. Bastide, F. Paterno. *Formal Specification as a Tool for Objective Assessment of Safety-Critical Interactive Systems*, in *Human-Computer Interaction – INTERACT'97*, July 1997, p.323-331, Sydney, Australia: Chapman & Hall.

[150] Palmiter, S., G. Lynch, S. Lewis and M. Stempski *Breaking away from the conventional "usability lab": the Customer-Centred Design Group at Tektronix, Inc.* Behaviour and Information Technology, 1994, **13**(1,2):128-131.

[151] Paterno, F. *A methodology to design interactive systems based on Interactors*, ESPRIT BRA 7040 Amodeus 2 Technical Report WP7, February 1993.

[152] Paterno, F. and M. Mezzanotte. *Formal Analysis of User and System Interactions in the CERD Case Study* in *Proceedings of EHCI'95, IFIP Working Conference on Engineering for Human-Computer Interaction,* August 1995, p.213-226, Wyoming, USA: Chapman & Hall.

[153] Paterno, F., C. Mancini and S. Meniconi. *ConcurTaskTrees: a diagrammatic notation for specifying task models*, in *Human-Computer Interaction – Interact '97*, July 1997, p.362-370, Syndey, Australia: Chapman & Hall.

[154] Pecheur, C. *Advanced Modelling and Verification Techniques Applied to a Cluster File System*, INRIA Research report 3416, May 1998.

[155] Peterson, J., P. Hudak, and C. Elliott. *Lambda in Motion: Controlling Robots with Haskell.* in *Proceedings of PADL 1999.* 1999. p. 91-105. Lecture Notes in Computer Science, 1551: Springer-Verlag.

[156] Peyton Jones, S.L. and P. Wadler, *Imperative functional programming.* in *Proceedings of ACM Conference on the Principles of Programming Languages: POPL'93*, January 1993, p.71-84: ACM Press993.

[157] Peyton Jones, S.L, A. Gordon and S. Finne. *Concurrent Haskell.* in *Proceedings of ACM Conference on the Principles of Programming Languages: POPL'96*, January 1996, p295-308. St. Petersburg Beach, Florida: ACM Press..

[158] Peyton Jones, S.L. et al, *The Haskell 98 Report*, http://www.haskell.org

[159] Peyton Jones, S.L., S. Marlow, and C. Elliott. *Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell.* in *Proceedings of 11th International Workshop on the Impelementation of Functional Languages: IFL'99.* September 1997. p. 37-58. Nijmegen, Netherlands. To appear in Springer Verlag's Lecture Notes in Computer Science Series.

[160] Peyton Jones, S.L. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, to appear in the 2000 Marktoberdorf Summer school.

[161] Pfaff G.E. (ed), *User Interface Managements Systems: proceedings of the Workshop on User Interface Systems held in Seeheim*, p. 125-133. Springer, Berlin, 1985.

[162] Puerta, A. and J. Eisenstein, *Interactively Mapping Task Models to Interfaces in MOBI-D*, in in *Draft Proceedings of Design, Specification and Verification of Interactive Systems'98.* 1998. Abingdon, UK.

[163] Ramage, M. *Developing a methodology for the evaluation of cooperative systems*, in *Proceedings of IRIS (Information Systems Research in Scandinavia)*, August 1997, Norway.

[164] Rational Software Corporation. *Unified Modeling Language Notation Guide*, version 1.0, 1997, Rational Software.

[165] Rogers, Y. *Exploring Obstacles: Integrating CSCW in Evolving Organisations*, in *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work.* 1994. p. 67-77: ACM Press.

[166] Ross S., Ramage M. and Rogers Y., *PETRA: Participatory Evaluation Through Redesign and Analysis* Interacting with Computers, **7**(4):335-360.

[167] Rowley, D and D. Rhoades. *The Cognitive Jogthrough: A Fast Paced User Interface Evaluation Procedure*, in *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems.* 1992. p.389-395 : ACM Press.

[168] Rushby, J. *Using model checking to help discover mode confusions and other automation surprises*, to appear in Elsevier Reliability Engineering and System Safety Journal.

[169] Sage, M. and C.W. Johnson *Interacting with Haggis: Implementing Agent Based Specifications in a Functional Style*, in *Human-Computer Interaction – INTERACT'97*, July 1997, p.126-133, Sydney, Australia: Chapman & Hall.

[170] Sage, M. and C.W. Johnson *Interactors and Haggis: Executable specifications for interactive systems, Design, Specification and Verification of Interactive Systems'97.* 1997. p. 93-109. Granada, Spain: SpringerWienNewYork.

[171] Sage, M. and C.W. Johnson *Pragmatic Formal Design: A Case Study in Integrating Formal Methods into the HCI Development Cycle,* in in *Design, Specification and Verification of Interactive Systems'98.* 1998. p. 134-155. Abingdon, UK: SpringerWienNewYork.

[172] Sage M. and C.W. Johnson *A Declarative Prototyping Environment for the Development of Multi-User Safety Critical Systems,* in *Proceedings of International System Safety Conference 1999 (ISSC'99), August 1999.*

[173] Sage, M. and C.W. Johnson *Formally verified, Rapid Prototyping for Air Traffic Control,* to appear in Elsevier Reliability Engineering and System Safety Journal.

[174] Sage, M. *FranTk: A Declarative GUI language for Haskell,* in *Proceedings of Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'2000,* September 2000, p106-118, Montreal, Canada : ACM Press.

[175] Salzmann, M. and S.D. Rivers, *Smoke and mirrors: setting the stage for a successful usability test.* Behavior and Information technology, 1994, **13**: (1,2) : 9-16.

[176] Sarter, N and D. Woods *How in the world did I ever get into that mode?: Mode error and awareness in supervisory control,* Human Factors Journal, 1995, **37**: (1) : 5-19.

[177] Scholz, E. and B. Bokowski. *PIDGETS++ - a C++ framework unifying postscript pictures, gui objects, and lazy one-way constraints* in *Conference on the Technology of Object-Oriented Languages and Systems (TOOLS USA 96),* 1996, Santa Barbara, California: Prentice-Hall.

[178] Scholz, E. *Imperative Streams – A Monadic Combinator Library for Synchronous Programming,* in *Proceedings of Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP'99,* September 1999, p261-272 :ACM Press.

[179] Scholz, E. *A framework for programming interactive graphics in a functional programming language,* PhD thesis, 1998, Freien Universitat Berlin.

[180] Scrivener, S., S. Urquijo, H. Palmen, *Breakdown analysis* in *CSCW Requirements and Evaluation,* P.J. Thomas, Editor. 1996, Springer.

[181] Shapiro, D. *The limits of Ethnography: Combining Social Sciences for CSCW,* in *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work,* 1994, p417-428 : ACM Press.

[182] Sighireanu, M. and Mateescu, R. *Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS,* in *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design,* June 1997, Zagreb, Croatia.

[183] Shum, S *QOC Design Rationale Retrieval: A Cognitive Task Analysis & Design Implications,* Rank Xerox EuroPARC, 1993, Technical Report EPC-93-105.

[184] Buckingham Shum, S. A. Blandford, D. Duke, J. Good, J. May, F. Paterno and R. Young, *Multidisciplinary Modelling for User-Centred System Design: An Airtraffic Control Case Study* in *Proceedings of the HCI'96 Conference on People and Computers XI,* 1996, p 201-209 : Springer Verlag.

[185] Buckingham Shum, S *Analyzing the Usability of a Design Rationale Notation,* in [133].

[186] Smilowitz, E., M. Darnel and A. Benson, *Are we overlooking some usability testing methods? A comparison of lab, beta and forum tests.* Behavior and Information Technology, **13**:(1,2):184-190.

[187] Sommerville, I. *Software Engineering,* 4th ed, 1992, Reading, Mass: Addison-Wesley.

[188] Song, G. *Mixing Visual and Textual Programming in a Functional Language.* M.Sc. Thesis, Department of Computer Science, York University, May 1995

[189] Sparud, J. and C. Runciman *Tracing lazy functional computations with Redex trails,* in *Procedeedings of PLILP'97,* 1997, p291-308, Lecture Notes in Computer Science, 1292: Springer.

[190] Storrs G. and P. Windsor, *Rapid prototyping for requirements capture,* in *Proceedings of ATIS,* London 1992.

[191] Suchman, L. *Plans and Situated Actions: The Problem of Human-Computer Communication*, 1987, New York. Cambridge University Press.

[192] Suchman, L and R. Trigg *Understanding Practice: Video as a Medium for Reflection and Design*, in *Design at Work: Cooperative Design of Computer Systems.* J. Greenbaum and M. Kyng, Editors. p65-89, Hillsdale, NJ:Lawrence Erlbaum.

[193] Sufrin, B. and De Moor, O. *Modeless Structured Editing*, in *Proceedings or the Oxford-Microsoft symposium in Celebration of the work of Tony Hoare,* September 1999, Cornerstones in Computing Series, MacMillan, to appear 2000.

[194] Sun Microsystems, *The Java 2 Tutorial*, http://www.javasoft.com/

[195] Sutcliffe, A.G. *From User's Problems to Design Errors: Linking Evaluation to Improving Design Practice Practical Evaluation Methods for Improving a Prototype*, in *People and Computers VII: Proceedings of HCI'92*, p 117-134 : Cambridge University Press.

[196] Szczur, M. *Usability testing on a budget: a NASA usability test case study*. Behavior and Information technology, 1994, **13**: (1,2) : 106-118.

[197] Telford, E. *Developing a UAN Browser in ClockWorks: a Case Study of Incremental Development using the Clock Methodology*. York University Technical Report CS-96-03. York University, Canada, June 1996.

[198] Thomas, M. *The Story of Therac-25 in LOTOS*, High Integrity Systems Journal, 1994, **1**:(1): 3-16.

[199] Thompson, S. *A Functional Reactive Animation of a Lift using Fran*, Technical Report, TR5-98, Computing Laboratory, University of Kent, UK, May 1998.

[200] Thompson, S., H. Cameron, P. King, *Modelling Reactive Multimedia: Events and Behaviors.* To appear in the Journal of Multimedia Tools and Applications.

[201] Thompson, S. *The craft of functional programming*, 1999, Addison Wesley.

[202] Toronto Star. *Bugs mar new air control system.* 3 August 1992

[203] Twidale, M. T. Rodden and I. Sommerville, *The Designer's Notepad: Supporting and understanding cooperative design*, in *Proceedings of Proceedings of the Third European Conference on Computer-Supported Cooperative Work :ECSCW'93*, 1993, p93-108, Milan, Italy : Kluwer Academic Publishers.

[204] Vullinghs, T., Tuijnman, D. and Schulte, W. *Lightweight GUIs for functional programming*, in *Proceedings of Symposium on Programming Languages: Implementations, Logics and Programs*: *PLILP'95*, 1995, p341-356, Lecture Notes in Computer Science, 982, Springer Verlag.

[205] Vries, G.d., T.v. Gelderen, and F. Brigham, *Usability Laboratories at Philips: Supporting Research, Development, and Design for Consumer and Professional Products.* Behaviour and Information Technology, 1994. **13**(1,2): p. 119-127.

[206] Wan, Z. and P. Hudak, *Functional reactive programming from first principles* in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI'00)*, June 2000, p 242-252, Vancouver, BC : ACM Press.

[207] Wesson, J., G.d. Kock and P. Warren, *Designing for Usability: A Case Study*, in *Human-Computer Interaction – INTERACT'97*, July 1997, p.31-38, Sydney, Australia: Chapman & Hall.

[208] Wharton C., J. Rieman, C. Lewis and P. Polson *The cognitive walkthrough method: a practitioners guide*, in *Usability Inspection Methods*, in J. Nielsen and R. Mack, Editors, 1994, John Wiley, New York.

[209] Wilson, S., M. Bekker, P. Johnson and H. Johnson, *Helping and Hindering User Involvement – A tale of everyday design*, in *Proceedings of ACM CHI'97 Conference on Human Factors in Computing Systems*, 1997, p178-185 : ACM Press.

[210] Winograd, T. and F. Floris, *Understanding Computers and Cognition: A New Foundation for Design*, 1986, Ablex, Norwooed, NJ.

[211] Wright, P. and A. Monk *Evaluating for design.* in *People and Computers V: Proceedings of HCI'89*, 1989, p345-358 : Cambridge University Press.

[212] Wright, P. and A. Monk *Cooperative Evaluation – The York Manual.* University of York, York 1991.

[213] Zirkler, D. and D. Ballman, *Usability testing in a competitive market: lessons learned*, Behavior and Information Technology, **13**:(1,2):191-197.

[214] Zucker, J. *The Propositional μ-Calculus and its Use in Model Checking*, in *Lecture Notes in Computer Science 693: Functional Programming, Concurrency, Simulation and Automated Reasoning*, P.E. Lauer, Editor, 1992, Springer Verlag.