Declarative Development of Interactive Systems

T.C. Nicholas Graham

Department of Computer Science
York University
4700 Keele St.
Toronto, Ontario
CANADA M3J 1P3

Preface

Graphical user interfaces have become a standard feature of modern application programs. Because user interface design is still more of an art than a science, the development of these user interfaces is challenging. The most standard technique for creating user friendly interfaces is iterative refinement, where the user interface is first designed, tested with users, and then redesigned. At the same time, user interfaces have the special problems of supporting computation guided by the user (as implemented in direct manipulation interfaces), and of correctly implementing the often subtle interdependencies between user interface components.

In this dissertation, Graham shows how techniques from *declarative programming* can be used to simplify the process of user interface development. In the functional *Clock* programming language, the programmer describes the behaviour of a user interface, rather than the algorithms required to implement it. From a declarative specification, a *Clock* interpreter automatically derives an implementation of the user interface.

A traditional problem with languages as high-level as Clock is in describing their formal semantics. To solve this, Graham introduces the *Temporal Constraint Functional Programming (TCFP)* framework, in which the semantics of extended functional languages can be specified. Because of its basis in temporal logic, it is possible within TCFP to state and prove properties of not only the Clock language, but also of other languages based on extended functional programming.

The work described in this dissertation represents a significant step towards simplifying the process of user interface development. By showing that declarative approaches are both theoretically sound and practically implementable, the work demonstrates that the declarative development of user interfaces is a promising approach for the future.

Berlin, December 1994 Stefan Jähnichen

Author's Preface

This work is the published version of a Doctorate of Engineering dissertation from the Technical Univerity of Berlin. The work was examined by Prof. Dr.-Ing. Stefan Jähnichen and Prof. Dr. Ulrich Geske. The chair of the examination was Prof. Dr. A. Biedl. The work presented in this book was largely performed at the GMD *Forschungsstelle für Programmstrukturen* in Karlsruhe, Germany.

There are many people whose help and support were crucial to the completion of this work. Foremost among these is Stefan Jähnichen, who generously provided me with the opportunity of working at the GMD Karlsruhe, and the freedom to pursue the ideas presented here.

I would also like to thank my colleagues at the GMD, who were the main source of the institute's stimulating and supportive atmosphere. I would particularly like to thank Birgit Heinz, who suffered the greatest burden of having to put up with my German, the other members of the Phoenix project: Roland Dietrich, Hendrik Lock and our colleagues in London and Nijmegen, and Robert Gabriel and Gerd Kock. The latter stages of the work could not have been completed without the help of Birgitt Schmidt.

My students also deserve thanks for their invaluable work on the Clock system. These are: Herbert Damker, Stefan Hügel, Catherine Morton, Roy Nejabi, Joachim Schullerer, Gekun Song, and Tore Urnes.

Many thanks are also due to Walter and Ingrid Tichy, who opened their home to me when I first arrived in Germany, to Reidar Conradi for generously hosting my six month stay at NTH, in Trondheim, Norway, and to Jim Cordy for his moral support throughout the whole process.

I would like to thank my parents for their strong support from start to finish. Finally, I wish to thank my wife, Kjersti, for her encouragement, patience and quiet confidence that carried me through the thesis writing.

Toronto, January 1995 — T.C. Nicholas Graham

Contents

Chapter 1

Introduction

A recent industry survey has determined that user interface development accounts for approximately 50% of the cost of producing modern application programs [76]. Reacting to this cost, much research effort has been devoted to making user interfaces cheaper to produce, maintain and modify. One promising approach is the *declarative* programming of user interfaces, based on techniques from functional programming [75, 64, 49, 53, 37, 111]. Functional languages provide a high-level programming model, suitable for the problems of rapid prototyping, testing and modification of user interfaces.

Functional programming of user interfaces faces the central paradox, however, that pure functional programs may not contain I/O. In order to support flexible, graphical interaction, most user interface languages mix functional and imperative constructs, thus sacrificing the high-level declarative programming model. Other approaches [79] provide only restricted I/O facilities, preserving the declarative programming model, but at considerable cost to ease and flexibility of programming.

This thesis proposes that despite this apparent paradox, it is possible for purely declarative programming languages to support the development of graphical user interfaces, while still providing the flexibility and ease of programming of imperative I/O systems. The *Clock* language is introduced to substantiate this claim: Clock combines the functional and object-oriented paradigms, while providing a high-level, purely declarative graphical I/O system. Clock is based on the *Temporal Constraint Functional Programming* (*TCFP*) framework, a novel framework for developing and reasoning about extended functional languages. Because TCFP is more flexible than traditional frameworks, it is possible to reason about Clock's I/O system, and to prove that Clock is indeed declarative while still supporting the flexible forms of I/O found in more imperative approaches.

1.1 Declarative User Interface Programming

One of the prime reasons that interactive software is expensive to develop is that user interfaces cannot be designed a priori, but must be developed experimentally through

iterative user testing and refinement [106]. Therefore, much of the research in user interface construction has focused on providing tools that permit rapid prototyping and modification of user interfaces.

One promising approach to helping with rapid prototyping of user interfaces has been functional programming. Functional languages support a high-level, *declarative* style of programming, where programmers specify *what* a program is supposed to do, rather than *how* it is to be done. In addition to this high-level programming model, modern functional languages also support rapid prototyping by providing high-level constructs such as built-in lists, higher-order functions, and polymorphic typing.

Pure functional languages have the problem, however, that they do not conveniently support input/output or non-determinism. Input/output constructs are the cornerstone of specifying interaction with the user, while non-determinism is required to model the dialogue-dominant nature of direct-manipulation user interfaces. The λ-calculus, upon which functional programming is based [54], has no means of specifying input/output or non-deterministic behaviour. It is because of these difficulties that many user interface languages introduce *impure* extensions to functional languages, such as imperative I/O constructs, variables, and assignments. Introducing such impure extensions sacrifices the declarative programming model of the language – in order to understand the ordering of assignments or I/O events, programmers must be aware of how programs are executed, therefore viewing their programs as algorithms rather than specifications.

To avoid sacrificing their declarative programming model, pure functional languages take the approach of banning I/O from the language itself, and allowing access to I/O functionality through a restricted external interface. Such approaches are often based on continuations [57] or monads [113, 114]. While preserving the declarative programming model, these approaches share two serious problems:

- I/O is the foundation of user interface construction. If I/O is outside the scope of the language, then 50% of average programs will be outside the scope of the semantics and reasoning techniques of the language;
- These restricted I/O interfaces impose a particular program organization in order to satisfy mathematical properties of the language. This program organization is not necessarily the best way to organize interactive systems.

In chapter 2 of this thesis, we shall see that the traditional functional programming paradigm has proven too restrictive to conveniently support user interface construction. We therefore propose an extended framework, Temporal Constraint Functional Programming (or *TCFP*), in which these restrictions are eased. Under TCFP, the λ-calculus is augmented with constraints in a temporal logic. These constraints allow the expression of I/O, non-determinism, persistent state, and concurrency. Using TCFP as a framework, it is possible to define extended functional languages where I/O constructs are designed primarily for their ease of use, rather than as clumsy extensions to a too-rigid framework.

To demonstrate that realistic languages for the development of user interfaces can be based on declarative techniques, the Clock language is introduced. Clock is designed to conveniently support the development of the direct-manipulation style of user interface, while maintaining many of the interesting properties of functional languages. Clock has a graphical, object-framework style of architecture language, similar to the Smalltalk MVC model [59]. Architecture components are programmed in a functional style, using a syntax similar to Haskell [56]. Graphical displays are specified in a high-level view language, loosely based on RVL [43]. Clock has been implemented, and runs on Sun workstations.

Through its basis in TCFP, Clock has a precise mathematical semantics. The nature of the semantics allows many implementations, potentially including implementations supporting concurrency and multiple users. Clock's formal semantics allow properties of the language to be investigated. A central theorem of the thesis demonstrates that despite all of the language's imperative extensions, functions in Clock remain referentially transparent.

Clock is an example of just one language developed within the TCFP framework. While much effort has been made to make Clock practical and realistic, many other styles of languages are also possible.

We now give a brief overview of functional programming, following which, we introduce Clock, and give an overview of the TCFP framework upon which Clock is based.

1.2 Functional Programming

Functional programming refers to a style of programming based on mathematical functions; for a complete overview of functional programming, readers are referred to [54, 28]. Functional programs map a set of inputs to a set of outputs. A functional program for computing factorials is:[1]

```
fact 0 = 1.
fact n = n * (fact (n-1)).
```

Here the function *fact* is defined to map a natural number onto its factorial. The function is defined by two equations. Functions may not have side-effects, such as assigning values to variables, or performing I/O. Modern functional languages have a *non-strict* semantics. In this example:

```
let f x = 3 in
    f (1/0)
end let.
```

the expression "*f (1/0)*" evaluates to "3", following the definition that for all values of x, "$f\ x = 3$", even when x is undefined.

[1] All examples use the Clock syntax, which is similar to that of Haskell [56].

Expressions in functional languages have the property of *referential transparency*, meaning that each expression has a unique value regardless of the context or time of its execution. For example, the expression:

```
(f x) + (f x)
```

can always be rewritten as:

```
2 * (f x)
```

This property does not hold of imperative languages, where f may return different values each time it is evaluated.

It is because of this property of referential transparency that functional languages are *declarative*. The goal of declarative programming is to allow programmers to specify *what* a program is to do, rather than *how* it is to do it. A defining characteristic of declarative programming is that programmers need not be aware of the order in which programs are evaluated; i.e., declarative programs have no explicit control flow. Referential transparency guarantees that the value of an expression will be the same no matter what the order of its evaluation. This lack of explicit control flow provides a high-level programming model, where programmers are freed from worrying about the temporal aspects of programs.

Standard input/output constructs do not provide referential transparency. If we were to provide a function *read* that inputs a value provided by the user of the program, *read* might return different values each time it was called. When writing an expression such as "*read / read*" (representing division of two numbers read from the user), the programmer would need to know the order of evaluation of the expression to know the order in which the values are read from the user. Introducing unrestricted I/O operations in a functional language therefore requires the language to provide explicit control flow. Therefore, in order to support I/O, functional languages must either provide restricted, referentially transparent I/O facilities, or must give up declarative programming. As we shall see in chapter 2, both of these approaches have been widely adopted.

1.3 The Clock Language

Modern application software is increasingly based on highly-interactive graphical user interfaces. Large software companies are discovering that the provision of interactive, graphical user interfaces introduces new complexities into the software development process [100]. In particular, the following problems are widely cited [73]:

- User interfaces cannot be designed *a priori* and then implemented using traditional techniques. Problems with user interfaces can only be determined experimentally. The design of good user interfaces requires iterations of design and testing with a user community. This *iterative design* is highly expensive, and requires the support of good rapid-prototyping tools.

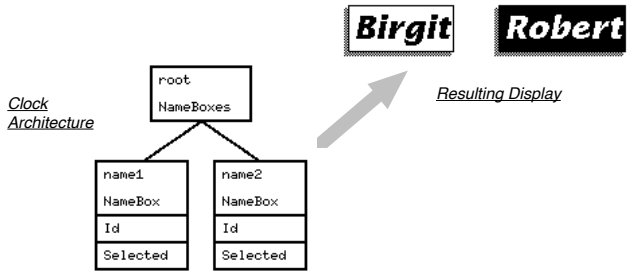


Figure 1.1: A Clock program displaying two "clickable" boxes.

- Modern user interfaces, particularly those based on the *direct manipulation* style [97], allow the user to direct the order in which tasks are performed. Tasks may be interleaved, or even performed concurrently. Application programs must therefore be capable of taking any input at any time.

- The behaviour and appearance of different components of a user interface are often related in subtle and intricate ways. A simple action in one part of a system may require updates of several other components. The problem of maintaining consistency between the different components of a user interface has been called the single most difficult problem in user interface programming [53].

The Clock language is designed to support the rapid-prototyping of interactive programs based on the graphical, direct-manipulation style. Clock is a purely declarative language: a graphical architecture language is used to specify the organization of Clock programs as a tree of components, while a functional language is used to specify the operation of the components themselves. The component approach allows existing user interface components to be reused in new architectures, allowing user interfaces to be rapidly constructed. The functional language used to program components allows simple specification of user interface views. Together, these formalisms provide a good basis for rapid prototyping.

To address the problem of consistency among components, a special class of functions called *invariant functions* are provided. These functions allow the programmer to give a declarative specification of what it is for a component to be in a good state. The Clock run-time system uses these functions to automatically maintain consistency among components.

The separation of the language into an architecture language and a component language is similar in approach to various object-frameworks suggested for user interface construction, such as Smalltalk MVC [59], ALV [48] and PAC [22]. The primary difference between the Clock approach and these object frameworks is that Clock's architecture language is built into the language, rather than being provided as a set of predefined

```
mouseButtonUpdt "Down" =
    if isSelected then
        deSelect
    else
        select
    end if .

mouseButtonUpdt _ = noUpdate.

invariant = noUpdate.

initially nm = all [setMyId nm, deSelect].
```

```
view =
    let msg =
          if isSelected then
              invert (paddedText 3 myId)
          else
              paddedText 3 myId
          end if
    in greyShadow (
          Box (
            Font hugeBoldItalicText msg
          )
        )
    end let.
```

Figure 1.2: The complete code implementing the *NameBox* component.

classes in a general object-oriented language. Knowledge of how to reason about architectures can then be built into the compiler, allowing powerful optimizations. Because of such optimizations as incremental updating of components and displays, Clock programs execute at close to production quality speed, despite the very high level language used to specify them.

Figure 1.1 shows an example of a Clock architecture and the display it generates. The program displays two boxes, each with a different name. Clicking on a box highlights it, and clicking on it again returns it to its normal state. In the architecture, the two components labeled *"NameBox"* are responsible for displaying the two boxes, and for handling the mouse input directed to the boxes. The component labeled *"NameBoxes"* combines the interactive displays of the two *"NameBox"* components, and creates a display containing both of them. The components *"Selected"* and *"Id"* are responsible for maintaining information about the name boxes to which they are attached, in particular, maintaining the name of the name box, and whether the box is currently highlighted or not.

Figure 1.2 gives the complete code implementing the *NameBox* component of the architecture. While the details of this functional code will be explained in chapter 4, this code serves to give a flavour of the language. The *view* function specifies that the display generated by the component is to consist of a box with a textual name. If the box is currently selected, the display view is to be inverted. In Clock, display views are a data type; because of this, interactive display views are actually first class values in the language, and the *view* function specifies how to construct such a value.

The *mouseButtonUpdt* function specifies how mouse clicks directed to this component are to be handled: in particular, if the mouse button is clicked down over the display of this component, the selection state of the component is to be switched. Requests such as *"isSelected"* are queries to the *Selected* component, while updates such as *"select"* modify the state of the *Selected* component. Despite this seemingly imperative behaviour, the language guarantees that the evaluation of any particular function will be referentially transparent. Referential transparency implies, for example, that during the evaluation of the view function, the request "*isSelected"* must always return the same value.

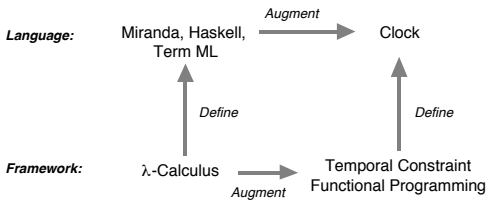


Figure 1.3: The TCFP framework extends the traditional λ-calculus, allowing the definition of interactive languages like Clock.

A prototype compiler and programming environment for Clock have been implemented. The graphical display of figure 1.1 was created by the ClockWorks programming environment [67]. The Clock system is still under active development: ongoing work is aimed at extending Clock to support the development of multi-user and multi-media applications.

1.4 The TCFP Framework

As a realistic and usable language, Clock demonstrates the practical use of declarative technology in the development of user interfaces. Clock, however, through introducing such imperative features as updates and requests and the manipulation of persistent state, can no longer be considered a pure functional language. Standard rules such as β-reduction and standard properties such as referential transparency can no longer be assumed to hold, but must be proven in the context of the new language.

To address these problems, we introduce a new semantic framework called *Temporal Constraint Functional Programming* (or *TCFP* for short.) Figure 1.3 shows the relationship between TCFP and functional programming. The λ-calculus forms a theoretical framework from which traditional functional languages such as Miranda, Haskell [56], and Term ML [63] can be developed, and where their properties can be discussed. By extending the λ-calculus with I/O primitives and constraints in a temporal logic, TCFP provides a framework allowing the development and investigation of extended functional languages.

TCFP is based on a *generate and restrict* paradigm. Under this paradigm, computations in an extended functional language may take on a set of different values, depending on their non-deterministic bahaviour. An extended functional program is said to *generate* a possible set of behaviours. Not all of these behaviours may be considered correct. Constraints in a temporal logic are then used to rule out incorrect behaviours, thereby *restricting* the possible behaviours generated by the extended functional program.

TCFP is in itself not intended to be a programming language, but rather a *framework* for the development of programming languages. As indicated by figure 1.3, imperative

extensions to functional languages can be specified as syntactic sugar for constructs in TCFP. TCFP therefore provides a basis for specifying the semantics of programming language constructs, and for reasoning about their properties.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 reviews current approaches to developing user interfaces. The description concentrates on declarative techniques, showing how existing techniques choose between impure extensions to functional programming, or restricted I/O constructs that preserve referential transparency. It is shown that the definition of *declarativeness* for user interface programming should go beyond simply considering referential transparency, and also consider higher level aspects of declarative support for consistency maintenance and view regeneration.

The third and fourth chapters describes the Clock language. It is shown how Clock provides the flexibility of imperative I/O systems, while providing a high-level, purely declarative programming style.

In order to demonstrate that Clock actually possesses the declarative properties discussed in chapters 3 and 4, it is necessary to define Clock's formal semantics. Chapter 5 introduces the TCFP framework, upon which the Clock semantics are based. It is shown how TCFP can be used to model concurrency, synchronization, variables, and shared memory. These mechanisms are then used to define the semantics of Clock in chapter 6.

Finally, chapter 7 discusses the properties of the Clock language. The notion of *declarativeness* is formally defined. Declarativeness *in the small* refers to the traditional properties of functional programming, particularly referential transparency. Declarativeness *in the large* refers to the declarativeness of Clock's constraint-based facilities for consistency maintenance and display updating. In this chapter, it is shown that Clock does indeed provide these declarative properties.

Chapter 2

Declarative Approaches to User Interface Development

It is widely accepted that declarative techniques are helpful in the development of user interfaces, as evidenced by the adoption of declarative technology in numerous commercial and research user interface development tools. This chapter motivates what the problems are in the development of user interfaces, and surveys solutions to these problems. The chapter particularly emphasizes solutions that adopt a declarative style, but is not limited to purely declarative approaches.

The survey is structured around the Seeheim model [93] of user interface management systems (UIMS's). While somewhat dated as an implementation model, the Seeheim model still serves as an interesting basis for classifying approaches to user interface development.

Tools based on declarative technology fall roughly into two groups: those developed by the HCI (Human-Computer Interaction) community, and those developed by the functional programming community.[1] In their work on user interface development tools, the HCI community has concentrated on functionality and ease of use, while the functional programming community has concentrated on pure approaches to supporting graphical I/O. We shall see that both of these approaches have benefits and drawbacks: the lack of formalism behind the UI tools can lead to semantic difficulties, while the pure functional approaches sacrifice ease of use to maintain their declarative properties.

This chapter begins with a general overview of the problems associated with user interface construction, followed by an introduction to the Seeheim UIMS model. Later sections then survey methods, languages and tools for user interface development, in the context of the layers of the Seeheim model. Finally, we discuss methods for specifying and reasoning about the semantics of declarative languages supporting interaction.

[1] While treading similar ground, these communities have largely not communicated. For example, two recent papers on declarative programming of user interfaces [75, 79] collectively refer to 62 other papers, yet do not have a single reference in common.

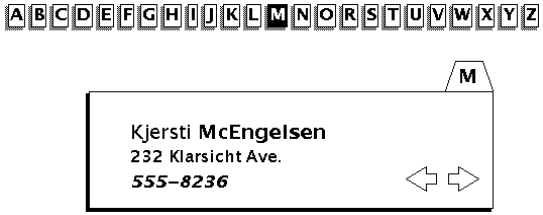


Figure 2.1: A card file name and address program implemented in Clock.

2.1 Problems in User Interface Construction

Moving from traditional batch software into the interactive world of graphical user interfaces has introduced many new problems in software construction. The characteristics of interactive software differ sufficiently from those of traditional software that our traditional development tools and languages are no longer adequate. In order to develop better tools, we must first understand the special characteristics of interactive software and how it is developed.

In describing these characteristics, we use an example program implementing a card-file name and address list (figure 2.1). The system allows the user to navigate through a set of cards, each of which gives a person's name, address and phone number. Clicking on a letter button moves to the first card at or following that letter. Clicking on the left/right arrow buttons moves to the previous or following card respectively. The various text fields can be edited. While simple, this example exhibits many of the problems in user interface development. Note that the Clock implementation of this card file is explored in detail in chapter 4.

Iterative Refinement: The traditional approach to software development is to design the software based on a set of requirements, and then implement it. The criteria for success are typically whether the software is correct, and how efficiently it runs on the target hardware. It is generally not expected that major redesign of the software should be necessary after implementation.

User interface design, however, requires user feedback. Limited approaches exist for evaluating user interfaces before they are implemented, such as cognitive walkthroughs [8] and the use of guidelines [107]. Ultimately, however, the only way to evaluate a user interface is to implement it, experiment with users, and modify the design based on user feedback. The technique of iterating between design, user testing, and redesign is called *iterative refinement*.

In an example such as the card file of figure 2.1, user testing could reveal important design errors – for example, the primitive navigation mechanisms provided

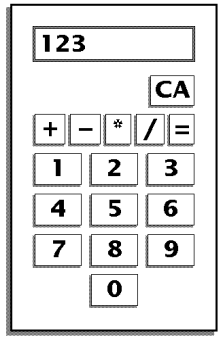


Figure 2.2: A calculator program implemented in Clock. This example was programmed by Gekun Song [101].

in the card file prove to be unwieldy for large data bases.

Important but less fundamental design errors can also be detected. Imagine a user is editing a name on a card, and moves to another card before committing the changes. It is unclear whether this action should be considered equivalent to committing the edit or aborting it, or perhaps should be forbidden altogether. Which of these decisions is correct (or at least acceptable) can be determined only through user testing. User interface design requires many such small decisions – user testing reveals which of these run counter to the user's intuition.

Iterative design is, however, expensive. User interfaces are difficult to build in the first place, and difficult to modify. For a programming language to support iterative design, the language must above all support easy creation and modification of user interfaces – the less time it costs to iterate, the more iterations will be performed. The wish to support iterative refinement has led to a proliferation of tools based on visual techniques for user interface specification, and prototyping tools based on high-level languages.

Direct Manipulation: The predominant interaction style used by modern user interfaces is called *direct manipulation*. In this approach, display objects such as menus, buttons and scroll bars are manipulated to cause actions in the underlying application. Usually, display objects give some kind of feedback to reveal application state. The name and address program of figure 2.1 is a direct manipulation interface based on the metaphor of a rolodex card file. Figure 2.2 shows a direct-manipulation calculator program. Figure 2.3 shows a direct-manipulation program implementing a university terminal reservation system.

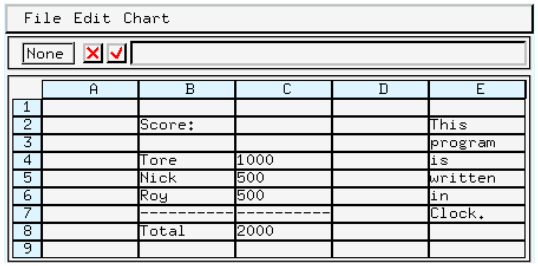


Figure 2.3: A simple spreadsheet. This example was programmed in Clock by Tore Urnes.

Direct manipulation interfaces have a property that the user controls the order in which interactions occur. Unlike application-dominant programs in which the user must respond to prompts from the program, in direct manipulation programs the user can initiate any of a set of *dialogues* with the application, and can move back and forth between ongoing dialogues at will. Programs supporting the direct manipulation style must therefore be capable of taking input intended for any dialogue at any time. Languages supporting the development of direct manipulation interfaces must be capable of handling inputs in non-deterministic orders.

Semantic Feedback: In order to allow ease of modification, it is usually considered beneficial to separate the user interface from the underlying application program. This goal is difficult to achieve, since parts of the user interface may depend on the state of the application program. These dependencies may affect low-level operation of the user interface, making abstraction difficult.

For example, the card file interface might forbid pushing a letter button for which there were no cards. This would require each button to be able to query whether cards were available at the time of its being pushed. Such use of application information in an interaction technique is called *semantic feedback*. In order to implement semantic feedback, user interface components must have fast access to application data, often making it difficult to separate application and user interface as cleanly as they should be.

Communication Among User Interface Components: In addition to semantic feedback, user interface components must be able to communicate with each other to perform their tasks. Different parts of a user interface can have subtle and complicated dependencies that require the updating of one component when another is modified. For example, in the card file, changing the surname of the current person may move the card's position alphabetically, in turn changing which of the letter buttons should be highlighted.

A traditional difficulty in user interface development is identifying all such dependencies between components, and correctly implementing them. Ideally, a user interface should be structured so that the most basic components such as scroll bars, menus and buttons are independent, and do not have built in knowledge of what other components are present in the same user interface. User interface languages should support the expression of intercomponent dependencies, while allowing clean structuring of individual components. It has been conjectured that maintenance of data consistency among user interface components is the single biggest problem in user interface development [53].

User Interface Consistency: One of the few fundamental guidelines in user interface design is that they should be consistent. Low-level consistency implies that interaction techniques such as scroll bars and menus should appear and behave in the same way throughout the user interface. High level consistency implies that within an interface, similar problems should be solved in similar ways – mechanisms for selecting, searching, or deleting should be consistent regardless of the context in which they are performed. Most platforms now come with style guidelines [2, 85, 104] specifying the behaviour of standard interaction techniques, and general rules for how interfaces should be designed. User interface tools should help in maintaining consistency within a user interface.

Concurrency: Many user interfaces support concurrent activities: a user interface may have a printer window displaying the printer's progress, while a fax window automatically reports on incoming faxes, while the user operates a word processor. Concurrent user interface processes may interact with one another – a compiler may insert error messages into a text editor as the user works on the program. Multiple users working on a common task can introduce concurrency – for example, distributed editors [58] or electronic chalkboards [102] allow multiple users to work concurrently on a shared data base that is reflected on each user's display.

Programming concurrency is in general difficult, and requires careful design and planning. It is therefore hard to combine support for rapid prototyping and concurrency within a single programming language. User interface tools must walk a difficult line of adequately supporting concurrency, while not sacrificing ease of programming.

From this list, it is clear that making functional languages suitable for user interface development goes beyond the problem of how to introduce I/O constructs into a functional language. The handling of problems such as semantic feedback, structured communication between components and concurrency must be solved by providing high-level mechanisms aiding program structure. Support for consistency in the user interface can be given by tools supporting reuse of components and designs. The support for rapid prototyping that is required to allow successful iterative design ultimately depends on how well rapid prototyping features are integrated with a system providing the high-level structure required by these other requirements.

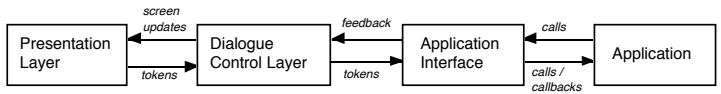


Figure 2.4: The Seeheim Model of User Interface Management Systems.

2.2 The Seeheim Model for User Interface Management Systems

We shall base our survey of existing solutions to the problem of building user interfaces on the Seeheim model of User Interface Management Systems [93]. As shown in figure 2.4, the Seeheim model splits tools for user interface development into four major layers: the *Presentation Layer*, the *Dialogue Control Layer*, the *Application Interface Layer*, and the *Application* itself.

The Seeheim model was originally proposed in 1983 as a model for the implementation of user interface tools, and a number of tools were actually built based on the model [99]. Since then, tools and toolkits have taken on more varied forms. The various tools tend to have strengths in some parts of the Seeheim model, and weaknesses in others. The model remains, therefore, a useful mechanism for understanding and classifying user interface tools.

In our survey of tools, we shall discuss each of the layers in the Seeheim model, and give examples of tools that aim to help at each layer.

The first layer in the Seeheim model is called the *presentation layer*. This layer is responsible for the physical appearance of the user interface, and for interpreting the most basic user inputs. The *presentation* of a user interface includes such aspects as the physical appearance of buttons, menus and text, and how they are arranged on the display. The presentation layer is also responsible for interpreting user inputs, such as mouse motion, clicking, and text entry, and passing it on to the dialogue layer.

The *dialogue layer* implements *dialogues* between the user and the application program. A simple example of a dialogue would be selecting an item from a menu. The selection process may consist of a number of primitive actions – moving the mouse to the menu bar, clicking the mouse down, moving the mouse to the correct item, and releasing the mouse button. At each point during this sequence of actions, the user interface gives feedback to the user indicating that the action has been correctly interpreted – moving the mouse causes a tracking symbol to move on the display; clicking on the menu causes the menu to pop up; moving between menu selections causes the currently selected menu item to be highlighted. This interchange between the user and the system, including the various feedback offered at each stage, is called a *dialogue*.

User interfaces must somehow be connected to an underlying application program.

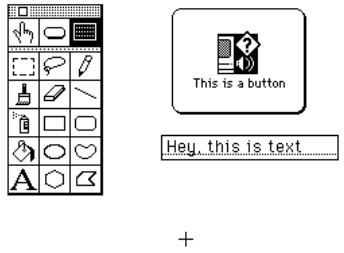


Figure 2.5: Building a User Interface with HyperCard: Programmers can select graphics and interaction techniques from a tools menu, and position and size them using direct manipulation.

This is accomplished via an *application interface* layer, that is capable of interpreting user actions, and invoking the appropriate part of the application to handle it.

The final layer is the *application program* itself. This layer implements the actual functionality of the program.

In the following sections, we shall explore a series of systems and approaches for user interface development. To give some structure to our discussion, we shall consider each approach within the context of the Seeheim model, indicating what sort of support for user interface development the system best provides. In the description, we shall concentrate as much as possible on declarative support for user interface development, to demonstrate that there is a long tradition within the user interface community of exploiting declarative description techniques.

2.3 Supporting User Interface Presentation

The *presentation layer* of a user interface specifies what appears on the display, and how basic interactions such as mouse clicks and keyboard entry should be handled. The output side of presentation specifies what basic display objects are present, together with such aspects as their colour and layout. The input aspect of the presentation layer represents the lexical structure of the interface, mapping raw inputs to higher level tokens.

2.3.1 User Interface Builders

While many tools still encourage textual programming of the presentation layer (e.g., raw Xlib programming [81]), modern tools tend to provide graphical interface builders

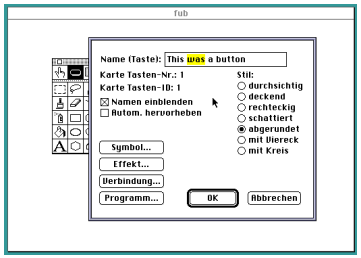


Figure 2.6: A HyperCard property sheet allows programmers to enter properties of interaction techniques, and to bind them to application code. The concept of property sheets is widely used in user interface builders.

to allow the specification of display objects, and their sizes and positions.

Figure 2.5 shows a simple example of building an interface in HyperCard [4]. The programmer selects from a palette of tools such as buttons, text fields and drawing tools, and literally draws the interface on a canvas (or *card* in HyperCard terminology.) The drawing that the programmer creates corresponds exactly to how the program will appear on the display as it is running. This direct manipulation approach to designing user interface presentation is far superior to writing code to generate presentation, since the programmer can immediately see how the interface appears, and quickly experiment to obtain an aesthetic presentation.

To establish properties of the user interface that cannot easily be specified with direct manipulation, the programmer double-clicks on the display object, and obtains a *property sheet* for the object (figure 2.6). The property sheet allows the programmer to specify, for instance, feedback effects on the display object, and how the object connects with computation.

Other example interface builders include the sophisticated NeXT Interface Builder [78], the FormsVBT form editor [6], Cardelli's pioneering work [12], and the G^2F system for generation of two-dimensional editors [32, 33].

Interface builders have limitations, however. The direct-manipulation style of an interface builder typically does not provide the full power of a programming language. This means that dynamic aspects of presentation often cannot be specified. For example, a tool for visualizing graph structures (such as Edge [88]) typically cannot be built with an interface builder, since there is no way of expressing a complex layout algorithm to position an arbitrary number of nodes in a graph. Typically, users of interface builders must eventually return to the world of textual programming for those dynamic parts of their user interfaces [25]. Some aspects of these problems have been addressed in experimental systems allowing programming by demonstration. These systems include Peridot [69, 68], a system for specifying interaction techniques by demonstration.

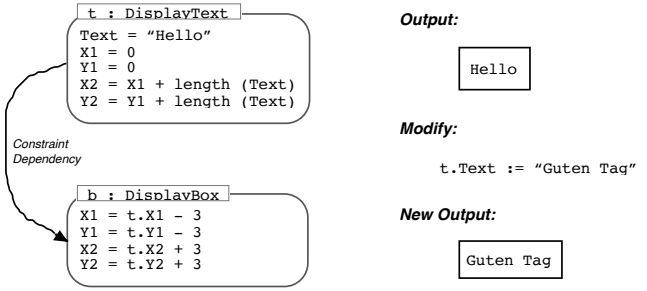


Figure 2.7: Constraints in a Garnet [70] or RendezVous [49] style of one-way constraint system. The objects created by a program specify a box surrounding a text. If the text is changed, the box is automatically resized to surround the new text with a border of 3 pixels.

User interface builders can be seen as purely declarative approaches to specifying presentation. Rather than writing imperative commands to draw pictures on a display, the programmer directly draws the pictures him/herself.

2.3.2 Constraints

One of the more tedious aspects of user interface construction is specifying the locations and sizes of primitives on a display. It is especially annoying that when one small part of a user interface's layout is changed, potentially the positions of many other objects may also have to be changed. A solution to this problem is the use of *geometric constraints* to specify relationships between the sizes and positions of display objects.

The use of constraints in user interfaces was first investigated in special purpose systems such as ThingLab [9], and are now available in many user interface toolkits and languages, such as Garnet [74], RendezVous [49] and Siri [53]. Even the Xlib [81] programming interface to the X Window System [95] includes a limited form of geometric constraints. Our presentation of constraints will be based on a simplified version of Garnet constraints. (The main simplification is to the syntax; Garnet is based on Common Lisp.)

Figure 2.7 shows a simple example of how two display objects can be linked with constraints. The object t represents text to be shown on the display. The lower-left corner of the text is the coordinate $(X1, Y1)$, set to be position $(0, 0)$. The upper-right corner is calculated from the size of the text. As shown in figure 2.7, the text is to be surrounded by a box, with a border of 3 pixels. A box object b is *constrained* to surround the text by linking the position of the box to the position of the text. For example, the $X2$ position of the box is set to being $t.X2+3$, or three pixels to the right of rightmost extent of the text. This constraint is automatically maintained, meaning that if the text is modified (e.g., to "Guten Tag", the German version of "Hello"), the

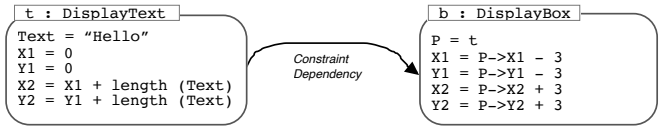


Figure 2.8: Constraints may involve *pointer variables*, providing indirection. Here, by setting the value "b.P", the box can set to surround an arbitrary object. (In this case, as in figure 2.7, the box is surrounding the text *"Hello"*.

box is automatically resized so that the constraint continues to hold.

Constraints such as these are called *one-way* constraints, since they only hold in one direction – if the size of the box is explicitly changed, the size of the text is not affected, and the constraint is effectively broken.

It is often useful to permit indirection in constraints, so that an object to which a constraint refers is determined at run-time. Figure 2.8 shows how the example from figure 2.7 can be recoded to use a *pointer variable* P to refer to the text node. This way, by reassigning $b.P$, the box can be set to enclose arbitrary objects. This indirection is crucial to the design of reusable user interface components.

Constraints need not only be used for maintaining geometric relations. They can also be used to maintain internal data consistency in a user interface. Figure 2.9 shows a Garnet-style approach to implementing the card file of figure 2.1. In this implementation, an object *cp* holds the data representing the current person. We assume that *cp* is somehow kept up to date as the user traverses the card file. Using constraints, the card display *c* can be automatically kept up to date as the card data changes. (Note that implementing this card display *c* would require several Garnet objects, not shown here.) An object *cl* uses a constraint to maintain the first letter of the last name of the current card. This allows the letter tag and letter buttons also to be automatically kept up to date. Constraints therefore provide a solution to the consistency maintenance problem, since programmers can specify consistency relations at a high level, leaving it up to the constraint solver to make consistency updates as necessary.

Evaluating the Constraint Approach

As pointed out by Vander Zanden, one-way constraints are actually functions [111]. Other languages implement *multi-way* constraints, where the directional aspects are not important; these languages include Siri [53, 52] and Constraint Imperative Programming [29]. A discussion of varying styles of constraints can be found in [41].

Constraint solving is in general very expensive. Most user interface constraint solvers, however, follow an incremental approach that is usually linear in the number of con-

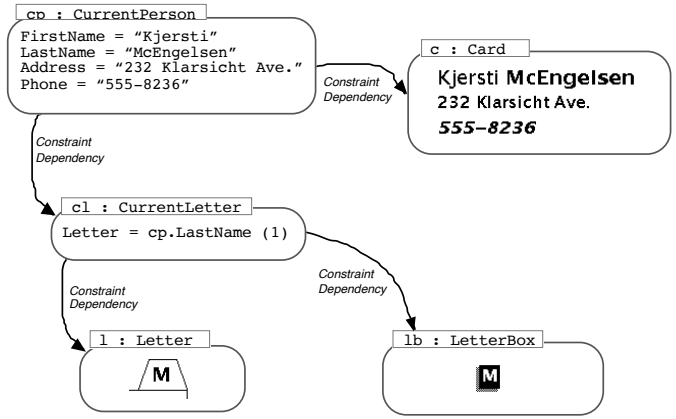


Figure 2.9: Constraints may be used to maintain consistency in user interfaces; this schematically shows a Garnet-style solution to the card file example of figure 2.1. Note that the code for the objects *Card*, *Letter* and *LetterBox* is not provided.

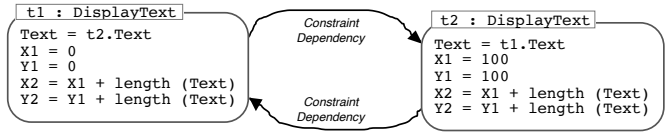


Figure 2.10: Example of a constraint loop – the text in both objects is constrained to be the same as the other. The semantics of this constraint depends on the constraint solver used.

```
b : DangerousBox

P = createInstance 'DangerousBox
X1 = P->X1 - 3
Y1 = P->Y1 - 3
X2 = P->X2 + 3
Y2 = P->Y2 + 3
```

Figure 2.11: Example of an infinite constraint sequence – a box surrounds another box, created as a side-effect of evaluation 'P'. These leads to an infinite sequence of *DangerousBox* objects being created.

straints to be solved [30]. Other examples of incremental constraint solvers include the RendezVous constraint system [49], and the Garnet constraint solver [112].

Constraints are essentially a declarative formalism, in that programmers should be able to specify consistency conditions and geometric constraints without explicitly coding how or when consistency updates are to be made. The aim of providing constraints in languages like Garnet is to permit programming in a "declarative style" [75]. The one-way constraints of Garnet and RendezVous have, however, two features which compromise their declarative nature: they permit *side-effects* and *constraint loops* in constraints.

A constraint contains a *side-effect* if its execution results in some observable effect other than the update of the object containing the constraint. For example, in Garnet and RendezVous, constraint side-effects are used to to create and destroy objects, implementing the dynamic aspects of user interfaces. In RendezVous, side-effects and pointer variables are combined to allow constraints to be modified as a result of their own execution [49]. This use of side-effects reveals the evaluation strategy of the constraint solver to the programmer – if a constraint has the side-effect of creating a new display object, the programmer must know exactly when the constraint is to be executed.

Figure 2.10 shows an example of a *constraint loop*. A programmer wishes to display two text strings, which are constrained to always contain the same text. As written, the constraint does not have a unique solution, so it is unclear what text should be displayed. This form of constraint loop can be detected at compile or run-time; however, when combined with side-effects and pointer variables, constraint loops can lead to unexpected behaviour, and difficult debugging. To debug constraint loops, a programmer must understand when constraints are evaluated, and in what order. This detracts from the declarative flavour of the constraints.

When combined, constraint loops, pointer variables and side-effects can lead to disaster. Figure 2.11 shows an example of an infinite constraint sequence. If an instance of *DangerousBox* is created, to resolve the boxes position, the evaluation of the *P* slot creates a new instance of *DangerousBox*. The new *DangerBox* will in turn create a new instance of *DangerousBox*, and so on. In general, this form of infinite constraint


```
tivoli<175> ftp ftp.cs.yorku.ca
Connected to wolf.cs.yorku.ca.
220-ftp.cs.yorku.ca FTP server (CS.YorkU.Ca 3.93) ready.
220-Report any problems to Sysadm@cs.yorku.ca.
220-Note: Anonymous FTP incoming requires setting "account"
220 Note: to appropriate email address for local user.
Name (ftp.cs.yorku.ca:graham): anonymous
331 Guest login ok, send email address ( eg "your.name@tivoli") as password.
Password:
230 Guest login ok, access restrictions apply.
ftp>
```

Figure 2.12: A human-computer dialogue establishing an *ftp* connection with the York University server. The user initiates an `ftp` command, and is prompted for a username and password.

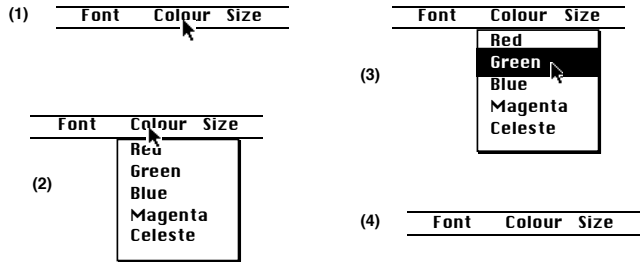


Figure 2.13: A *menu* dialogue: the user clicks on the menu bar (1), and the menu is shown (2); the user selects a colour (3), and releases the button (4).

sequence is not detectable, even at runtime.

2.4 Supporting Human-Computer Dialogues

A human-computer *dialogue* is a sequence of interactions in which a user engages in order to carry out some task. A dialogue normally involves an interchange of information, where the user receives feedback following his/her actions. For example, in figure 2.12, a user carries out a dialogue with the computer to establish an *ftp*-connection with a remote site. The user initiates the dialogue with an *ftp* command; the user is prompted for a username and a password, and is finally informed that the connection has been established. At each stage of the dialogue, the computer provides *feedback* indicating to the user that his/her input has been successfully interpreted. Another example of

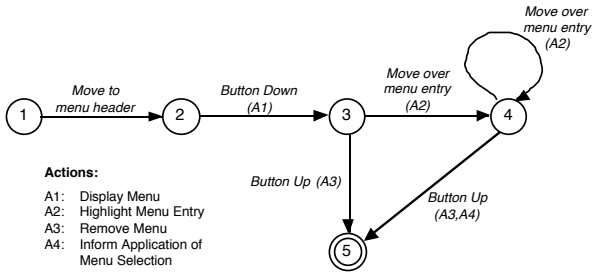


Figure 2.14: An augmented transition network encoding of the menu dialogue of figure 2.13. The arcs of the network are annotated with semantic actions to be carried out when the arc is traversed.

a dialogue (figure 2.13) is the sequence of steps required to make a selection from a menu – the user moves the mouse to the menu bar, clicks the mouse down, moves to the desired entry, and releases the mouse. At each stage, the system provides feedback indicating the currently selected menu item.

2.4.1 Specifying Dialogues with ATN's and Grammars

There has been considerable research into tools and formalisms for specifying legal dialogues, much of which is summarized by Green [44] and Olsen [83]. Under these approaches, the programmer provides a high-level dialogue specification, from which the dialogue control component of a user interface is automatically generated.

One form of dialogue specification is the Augmented Transition Network (ATN). ATN nodes represent user interface states, and arcs represent transitions caused by user actions. Figure 2.14 shows how an ATN can be used to encode the menu-selection dialogue of figure 2.13. Arcs are tagged with *semantic actions*, indicating what is to happen in response to the user's actions. The pure ATN approach has not been widely adopted, largely because ATN's rapidly become large and unmanageable. Practical ATN-based systems commonly use extensions to standard finite state machines, such as the use of external code to encode state, and meta-rules to guide the activation of arcs.

A second approach is the use of grammars, as in SYNGRAPH [84]. Grammars are similar to ATN's, including the use of semantic actions to indicate the effect of user actions. Figure 2.15 shows the menu-selection dialogue encoded as a grammar. The semantic actions attached to a production are triggered whenever the production is matched. The problem with grammars is in their implementation. If matching a production requires look-ahead, it is unclear whether intermediate semantic actions should be triggered or not. Semantic actions that involve feedback to the user are not reversable, and therefore cannot be performed until it is clear that they are really


```
selectFromMenu  ::=  moveToMenuHeader
                     ButtonDown A1
                     finishMenu

finishMenu      ::=  ButtonUp A3
                 |   { MoveOverMenuEntry A2 }+
                     ButtonUp A3 A4
```

Figure 2.15: A grammar encoding of the menu dialogue. The productions are annotated with semantic actions to be carried out when the production is matched; the semantic actions are shown in figure 2.14.

required. Look-ahead is not necessarily possible, since the lookahead may require actions that the user has not yet performed. In practice, the class of grammars that can be used to specify dialogues is restricted, and the grammar-writer must be aware of the restrictions. Attribute grammars [7, 94] have been used successfully as a more powerful approach to dialogue specification.

2.4.2 Event-Based Approaches

Both the grammar and ATN approaches perform poorly in the case of multi-threaded dialogues, since it is difficult to indicate the effect of interleaving actions in two or more dialogues. A better approach is production systems, which allow declarative specification of dialogues as a set of rules [82]. The Event-Response Language (ERL) used in the Sassafras User Interface Management System [47] uses a rule-based form of system to specify concurrent dialogues.

Programming languages supporting event-based programming include Esterel [16], Forms/3 [10], and Siri [53, 52]. Of these, both Forms/3 and Siri are declarative languages in which the reaction to events is specified in a functional and constraint-based style.

Concurrent Clean

The Concurrent Clean language [80, 110] provides a functional approach to I/O specification. Clean uses a systems-based model of input/output. A type *World* is used to encode the state of the environment at any given time. Input is therefore a mapping from a *World* (the "world" before the input) to a *World* (the "world" after the input.) The *World* parameter must therefore be passed to and returned by all functions in the program that either perform I/O, or call some other function that performs I/O.

As pointed out by Hudak and Sundaresh [55], the problem with the systems approach is that multiple, differing copies of the world may result, potentially leading to paradoxes. Clean avoids this problem simply by banning the creation of multiple copies through a special *UNQ* (or *unique*) type attribute applied to the *World* type.

Figure 2.16 shows an example of a systems-based I/O approach (using a Haskell-like

Predefined I/O functions:

```
read :: World -> (String,World).
write :: World -> String -> World.
```

Prompting program:

```
promptRespond :: World -> (String,World).
promptRespond w =
    let w' = write "Next name:" in
        read w'
    end let.

readNames :: World -> ([String],World).
readNames w =
    let
        (n,w') = promptRespond w,
    in
        if n = "Done" then
            ([], w'')
        else
            let (ns,w'') = readNames w' in
                (n:ns, w'')
            end let
        end if
    end let.
```

Sample Execution:

```
Next name:  Roy
Next name:  Catherine
Next name:  Tore
Next name:  Song
Next name:  Done
```

Figure 2.16: A systems-based I/O approach in the style of Concurrent Clean.

syntax). The function *readNames* prompts the user for a list of names, returning the names as a list of strings. Because of the need to present a *World* parameter to the I/O operations, the *promptRespond* and *readNames* functions must receive and return a *World* parameter themselves. It is clear from the example how quickly this becomes cumbersome.

Noble and Runciman have performed a significant case study with Clean [79].

2.4.3 Dialogue Combinators

A purely functional approach to dialogue specification is *dialogue combinators*, first pioneered by Thompson [108], and later extended for dynamic graphical user interfaces by Dwelly [26]. Under the dialogue combinator approach, a program is considered to map a list of inputs to a list of outputs. The list of inputs represents all inputs the user will ever perform; through lazy evaluation, this list is instantiated as the user performs inputs. Primitive I/O operations allow reading of inputs, and writing of (possibly graphical) outputs.

The dialogue combinators allow the combination of these primitive I/O operations in interesting ways. Combinators exist for sequencing, conditions, and repetition, allowing completely general specification of dialogues using only combinators. Programs written with dialogue combinators can, however, be difficult to read and understand.

Predefined I/O functions:

```
read :: String.
write :: String -> String.
```

Prompting program:

```
     nextEvent :: String.
E1:  nextEvent = write "Next name:".
E2:  nextEvent = read.

     events :: [String].
     events = nextEvent : events.
```

Sample Execution:

```
Next name:  Roy
Next name:  Catherine
Next name:  Tore
Next name:  Song
...
```

Constraints:

```
E1 -> T(E2), E2 -> T(E1).
```

Figure 2.17: A SIAN program implementing a simple dialogue reading a (potentially infinite) list of names. The prompting and input are interleaved using a temporal constraint.

2.4.4 SIAN

Another interesting approach allowing the programming of dialogues in a functional language is Darlington and While's SIAN language [24]. In this approach, the Hope language is extended with constructs permitting non-determinism and I/O. Although the approach is no longer purely functional (functions are not referentially transparent due to the presence of non-determinism and I/O), SIAN can still be considered to be declarative, since no control structure is imposed to order the program's evaluation.

Because of this lack of control structure, programs written in the basic SIAN language could have I/O occuring in any jumbled manner. In order to give the programmer some control over the order of I/O activities, a constraint language is provided as a meta-language, allowing control over the reduction order used to evaluate programs.

Figure 2.17 gives an example of a very simple SIAN program using this approach. (We use a Haskell-like syntax for the example rather than SIAN's own Hope-like syntax.) A simpler version of the program of figure 2.16, this program repeatedly prompts the user for a name, and returns the infinite list of names read. The core of the program is the function *nextEvent*, which is a non-deterministic function defined using two equations ($E1$ and $E2$.) When invoked, *nextEvent* will, at random, either prompt the user with *"Next name:"* or will read a string value. A constraint is used to specify that prompting and reading are to operate in lock-step. The constraint specifies that if the current evaluation of *nextEvent* uses equation $E1$ (i.e., if $E1$ holds), then $E2$ should be evaluated next time (i.e., "tomorrow (E2)", or "T(E2)" holds). Similarly, if $E2$ is used this time, then $E1$ should be evaluated next time.

While SIAN constraints appear to be low-level for actually programming user interfaces, SIAN provided much of the inspiration for the TCFP framework for defining and reasoning about I/O in functional languages [42, 40].

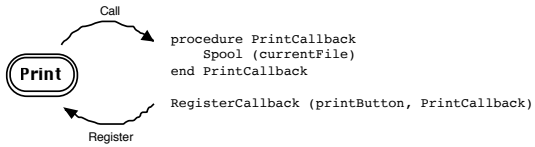


Figure 2.18: Example callback structure. An application registers a callback procedure (*PrintCallback*) with a button (*PrintButton*). When the button is pushed, *PrintCallback* will be called.

2.5 Connecting Applications to User Interfaces

User interfaces must somehow be connected to the application code implementing their functionality. This section reviews some approaches to how this is done. We first consider low-level connection approaches, concentrating on the traditional approaches of *callbacks* and *continuation functions*. We then review some more structured approaches, especially including user interface architectures.

2.5.1 Callbacks

The most standard mechanism for connecting user interface code to applications is the *callback* procedure. Callbacks are used in most user interface toolkits, such as the X Window System [95], OSF Motif [85], the Macintosh Toolkit [3], SUIT [89], Garnet [74], and InterViews [62].

Consider that we are developing a system in which a *print* button is used to activate a printing facility. Toolkit libraries typically provide a button widget, which can be instantiated to appear with a "Print" label at the desired screen location. We wish to specify that when the button is clicked, a *Spool* procedure is to be invoked with a variable representing a file to be printed. To do this, we create the callback procedure *PrintCallback*. We register this callback with the button, so that whenever the button is clicked, *PrintCallback* is invoked.

In toolkits, widgets commonly use numerous callback procedures to allow them to be used in arbitrary contexts. Programmers have have to learn many of the possible callbacks a widget can generate before being able to use it. The large number of callbacks and possible contexts in which they can be invoked can lead to a "spaghetti" style of programming, hindering program clarity and maintainability [71].

2.5.2 Continuations

The *continuation passing style* of programming provides application interface support for functional programming on a similar level to the callback approach used in imperative languages. Continuations were first pioneered in the Nebula operating systems


```
data IORequest = Read (Num -> IORequest)
               | Write Num IORequest
               | Done.

eval Read
     fn v1 ->
          Read
               fn v2 ->
                    Write (v1 + v2) Done
               end fn
     end fn.
```

c1 c2 c3

Figure 2.19: A simple continuation-based I/O system, and a program that reads in two numbers and writes out their sum.

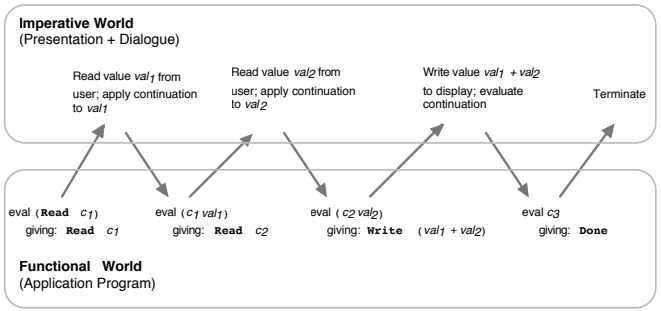


Figure 2.20: Execution of the continuation-based example of figure 2.19. The execution consists of a sequence of pure functional computations interleaved with the imperative actions of the I/O requests.

project [57], and have now been adopted as fundamental I/O facilities in LML [5], Haskell [55, 56], Opal [90], and Hope [91, 92].

The notion behind continuations is that pure functional computations cannot perform I/O directly while maintaining referential transparency. The notion of programs is therefore redefined to mean a *set* of pure functional computations, interleaved with which a set of imperative operations such as I/O may occur.

Figure 2.19 shows an example of a very simple continuation-based I/O system. Each functional computation must be of type *IORequest*, a data type defined to have three possible values. The *Read* I/O request indicates that a number is to be read, and that the argument should be applied to that value, giving a new I/O request. The *Write* I/O request indicates that the given number is to be written, and then the given expression of type *IORequest* is to be evaluated. Finally, the *Done* request indicates that no more computation is to be performed, i.e., that the program should terminate.

The example program in figure 2.19 uses continuation-based I/O to read in two numbers from the user, and to write their sum on the display. Figure 2.20 shows how this

program is executed. First, the main expression is evaluated to the request *Read* c_1, indicating that a value is to be read, and c_1 is to be applied to the value. The second functional computation consists of applying the continuation function c_1 to the value read provided by the user, which we call val_1. This expression is reduced to a new *Read* request, which is again evaluated. Finally, the *Write* request is performed, and the program terminates with a *Done* request.

This I/O request and continuation-based interface forms the *application interface* of the Seeheim model. The presentation and dialogue control components are performed in the imperative world of the I/O requests. The continuation functions are a form of call-back, used to communicate the results of I/O activity back to the functional world. The I/O requests are responsible for performing the actual I/O and sequencing I/O activities, while the functional code specifies whatever computation needs to be performed.

In practical languages, I/O requests have been defined for many forms of I/O, including interprocess communication, and as an interface to graphics toolkits. An extreme example of the use of I/O requests as an application interface is Sinclair's Haskell-Tcl/Tk interface [98], where user interfaces are programmed in the imperative Tcl/Tk language [86], while applications are programmed in Haskell.

In practice, the continuation passing style is sufficiently expressive to allow the construction of user interfaces. However, as can be seen in the example of figure 2.19, the style is cumbersome and unintuitive. It is difficult to convince imperative programmers of the benefits of functional programming when they have to write such code for as simple an activity as reading two numbers and printing their sum.

A similar effect to continuations can be achieved through the use of *monads* [113, 114]. While the underlying formalism is different (and more general), the style of programs is similar, and prone to the same usability problems as the continuation-passing style. Another related approach is *token streams* [103], which are also present in Haskell [55]. While elegant, token streams have proved difficult to use in practice.

In later sections, we shall see that many of the more elaborate approaches to pure functional I/O are in effect disguised versions of CPS, where programmers are to varying degrees shielded from the low-level details of the continuation passing style.

2.5.3 Structured Approaches to the Application Interface

A number of more structured approaches exist for organizing how applications are connected to user interfaces. We shall first consider approaches developed within the HCI community, then approaches for pure functional languages. Many of the HCI approaches are applicable to both declarative or imperative languages. These approaches mainly relax the Seeheim assumption that there is a single user interface component attached to a single application component, instead allowing more general connection structures.

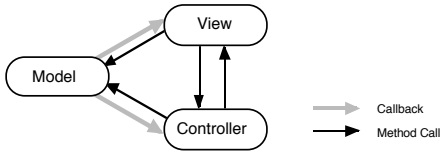


Figure 2.21: The Model-View-Controller Methodology. A *Model* encodes the underlying application. The *View* maintains the display view. The *Controller* handles input. (Diagram adapted from [48].)

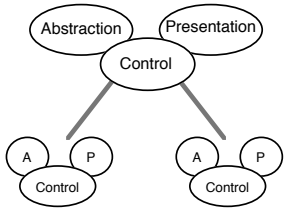


Figure 2.22: The PAC Model. An interaction technique is divided into *presentation*, *abstraction* and *control* components. These components can be arbitrarily nested.

The MVC Methodology

One of the earliest and most influential is the Smalltalk Model-View-Controller (MVC) methodology [59]. In MVC, interactive systems are divided into three components (figure 2.21). A *model* is used to encode the application functionality. A *model* is an abstract representation of the data underlying what appears on the display. The model is connected to a *view*, which is responsible for maintaining the display, and a *controller*, which is responsible for handling input. The model, view and controller communicate via a combination of method calls and callbacks. When the model is changed, it can notify the view and controller via callbacks that a change has taken place. Each is then responsible for bringing its own state up to date. Direct communication with the model is possible via method calls.

MVC is more structured than raw callbacks, since a single, clear communication strategy is used for all parts of the system. This helps somewhat to untangle the callback spaghetti.

The PAC Model

Another system similar to MVC is the Presentation-Abstraction-Control approach [20, 21, 22]. As shown in figure 2.22, PAC programs are organized into groups of three

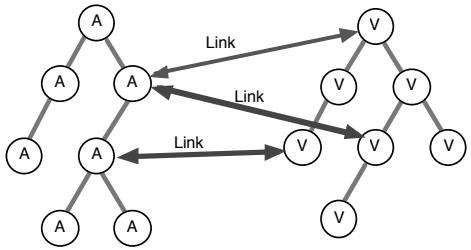


Figure 2.23: The ALV Model. Abstractions and views can be grouped into tree structures connected by constraints; constraint links are used to connect abstractions and views, and to automatically maintain consistency between them.

components. An *abstraction* implements the data underlying the view; this component corresponds to an MVC *model*. The *presentation* implements an interactive display view, implementing both the screen appearance and its interactive behaviour. The *control* is responsible for implementing consistency maintenance between the abstraction and presentation. In our Seeheim terminology, the presentation corresponds to the presentation and dialogue control layers; the abstraction corresponds to the application, and the control corresponds to the application interface.

In PAC, interactive applications can be structured as a tree of PAC clusters. Each subtree implements an interaction technique; these interaction techniques can then be grouped to provide more powerful interaction techniques, or a complete application. This tree-like structure represents an important advance over earlier (and many later) systems, in that the programmer is not forced to divide the entire program arbitrarily into application and user interface; the tree instead allows interactive systems to be built through layered abstractions.

One difficulty with the PAC model is that it is unclear how components on different levels of the tree are to communication, as is required to implement semantic feedback or consistency constraints.

The ALV Model

Another model similar to PAC is the *Abstraction-Link-View* (ALV) model [48], as implemented in the RendezVous language for development of multi-user applications [50, 87].

As shown in figure 2.23, ALV programs are divided into an *abstraction* and a *view*, corresponding to PAC's abstraction and presentation respectively. Both the abstraction and view are specified as hierarchies of components, using constraints to specify consistency maintenance. ALV *links* are used to connect the abstraction and the view. Links are special constraints allowing the view to be kept up to date with changes in the abstraction, and vice versa. Unlike PAC, where the control is used to keep abstraction

and presentation of individual components up to date, ALV links may be connected arbitrarily between any number of abstraction and view components.

In Hill's RendezVous implementation of ALV, constraints are similar to Garnet constraints, but with enhanced facilities for indirection and for side-effects [49]. Hill claims that side effects are an essential part of programming with constraints. Although powerful, side effects are clearly not declarative, and expose the frequency and order in which constraints are evaluated.

The Relational View Model

Similar to the ALV model is the Relational View Model. In this model, applications and user interfaces are specified separately. A *relation* is specified, indicating a consistency condition between the two. Whenever the application data changes (through program execution), the UI state will be automatically updated to reflect the change; whenever the UI state changes (through user input), the application data is automatically updated to reflect the change. This organization supports *mixed control* user interfaces [45], where control is shared between the application program and the user.

The Trip-2 system [105] implements relational views using Prolog – the application and user interface are programmed separately, and a prolog relation is used to connect them.

Weasel [43, 109] uses the Graphical View Language (GVL) [17] to implement the Relational View Model. Under this approach, the user interface is inferred from the relation, meaning that user does not have to specify the user interface directly, as would be the case in the ALV model.

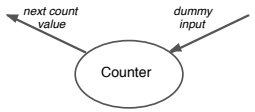
The Relational View Model is described in more detail in chapter 4.

Fudgets

The *Fudget* approach [13, 14] also encourages the structuring of interactive systems as a hierarchy of components. Components (or *fudgets*) are connected in a data-flow model, where the output of some component becomes the input of another. Fudgets may be connected via a set of fudget combinators, or may be grouped to form composite fudgets. A library of fudgets implements the basic interaction techniques found in most toolkits, such as buttons, menus and scroll bars. These fudgets communicate directly with a windowing system, and provide a high-level, message passing interface to the fudget programmer.

Most aspects of user interface presentation are built in to the predefined fudgets, although programmers have control over layout. More complex dialogues can be built into "abstract" fudgets, which may combine any number of presentation-level fudgets. The application program is also typically an abstract fudget.

Fudgets themselves are programmed using a continuation passing style – either using I/O continuations to implement the interface to the windowing system, or using special *stream processing* requests to implement application functionality. Figure 2.24 shows




```
data SP α β = GetSP (α -> SP α β)
            | PutSP β ([α] -> SP α β).

counter =
    let
        count n =
            GetSP
                fn _ ->
                    PutSP [n+1] (count (n+1))
                end fn
    in
        count 0
    end let.
```

Figure 2.24: An example abstract fudget implementing a counter, and the "Stream Processing" code implementing its functionality. Example adapted from [14].

an example of how fudgets are programmed. The example implements a counter – whenever the counter receives an input, the input is discarded, and the next counter value is written out. Two requests, *PutSP* and *GetSP* are used to read and write values to the fudget's I/O ports; the types of the requests indicate that a fudget should take values of type α on its input port, and β on its output port. The counter function generates a sequence of requests, repeatedly reading an input value and writing the next counter value.

The Fudget approach is a step up from programming with raw continuations, but has a number of difficulties. The basic model for component connection is that a component should read from one fudget, and write to one fudget. This restricted organization makes it difficult for a fudget to notify several user interface components of a state change; as we have seen in earlier examples, such complex data dependencies are common in user interfaces. Consistency maintenance is also made challenging by the message passing communication of fudgets. There is no high-level mechanism for specifying consistency constraints between fudgets, so all consistency conditions must be programmed by hand.

Fudgets implicitly carry a state with them. Rather than having state represented explicitly in a variable, state is carried as the continuation parameter in stream processing requests. Since there is no way for fudgets to share state, each fudget must represent all state it might possibly need. This inevitably leads to multiple representations of the same data, worsening the already difficult problems of state maintenance.

Finally, although programmers are somewhat shielded from using continuations, the fudget programmer must ultimately be comfortable with the CPS programming style.

A similar approach to Fudgets is the combination of functional programming and data flow embodied in Prograph [64, 38, 61]. In Prograph, objects are connected together using a visual editor. An object's 'methods' are programmed in an impure functional language, in which persistent data and side effects are used to avoid the need for continuation-based programming. Objects may have arbitrary input and output ports, and flexible object connection and data multiplexing allows far simpler design than is possible with the Fudget combinators.

2.6 Support for Reasoning about Languages for Developing User Interfaces

The functional programming community has placed great importance on the mathematical foundations of functional languages, and on being able to reason about the properties of functional programs. Despite this, support for user interface development is still undergoing sufficient investigation that the formalism has yet to catch up to the practice. The HCI community, on the other hand, has tended to place more value on the usability and application of their languages and tools than on their formal underpinnings. Because of this, there is relatively little in the way of developed formalisms for discussing and reasoning about interaction in declarative languages.

Part of the problem with trying to formalize the support for user interfaces in traditional functional languages is that the languages do not really support user interface development at all. As was seen in section 2.5.2, most functional languages provide support on the level of the application interface, and then leave the actual structuring of user interfaces out of the language.

Andrew Gordon addresses this problem in his framework for reasoning about I/O in functional languages [39]. This framework is based on the π-calculus [65], a formalism combining the λ-calculus and a CCS-style calculus [66] for reasoning about processes. Gordon uses the λ-calculus embedded in the π-calculus to define functional languages, while the process calculus is used, for example, to describe the meanings of continuations and how continuations interact with one another. Perry [92] uses Gordon's approach to specify the meaning of a continuation-based concurrent functional language.

Thompson [108] uses a trace-based approach to define the semantics of dialogue combinators, and to demonstrate some of their important properties.

Other more traditional approaches remain possible for describing the semantics of languages involving concurrency and interaction, such as Hoare's CSP [51], or power-domain based denotational semantics [96].

A number of the languages developed specifically for user interface development are based on mathematical formalisms: Garnet constraints are based on standard algebra; ATN and grammar-based methods for dialogue specification are based on language theory. However, the presence of side-effects and restrictions that expose the computational model of these approaches can invalidate the underlying formalism.

2.7 Conclusion

This chapter has reviewed the problems of programming user interfaces, and has surveyed a wide variety of declarative or near-declarative approaches to supporting user interface development. From this literature review, a number of points should be clear.

First, user interface programming is not a simple extension of traditional programming.

The demands of direct manipulation interfaces require special support, particularly in the areas of responding to asynchronous inputs, concurrency, and consistency maintenance. User interface programming requires support in many different areas, from presentation, to dialogue management, to organizing the interface between user interface and application. Therefore, simply providing an interface builder to aid in programming presentation, or adding I/O continuations to a functional language, is not sufficient to solve the problems of user interface programming.

Second, ease of use is paramount in user interface languages. The demands of rapid prototyping and iterative refinement mean that programmers will not willingly put up with clumsy I/O systems. Practically, this means that languages whose I/O is based on continuations or monads have no realistic chance of widespread acceptance.

Third, the notion that declarative styles of specification are useful in user interface development is widely accepted, both in the user interface and functional programming communities. What is still at issue is how the tradeoff between ease of use and mathematical purity should be resolved. Purity seems to lead to clumsy I/O systems. Impure constructs such as side-effects and pointer variables expose the underlying implementation, and can lead to disasterous effects, such as infinite constraint sequences.

These three points form the philosophical basis of the work with the Clock language that is reported in this dissertation. As will be seen in chapter 4, Clock adopts many ideas from existing user interface languages, such as the use of high-level declarative constraints for consistency maintenance, the use of an architecture language for program structuring, and the adoption of a simple I/O system. At the same time, Clock provides the most basic properties of pure functional programming, most importantly, referential transparency of expressions. Clock also solves the constraint loop problem; the design of Clock means that it is impossible to write constraint loops or infinite constraint sequences. Clock achieves these goals through its basis in the TCFP formalism. Chapter 6 presents the Clock semantics in terms of TCFP, while chapter 7 demonstrates that Clock provides the properties of referential transparency, and freedom from constraint loops.

Chapter 3

Overview of the Clock Language

Clock is a high-level language intended for the rapid prototyping of interactive systems. Clock is designed to support easy creation and modification of user interfaces, while supporting convenient reuse of existing user interface components. Clock programmers are freed from many of the low-level details of creating user interfaces, particularly through:

- A high-level view language for specifying interactive display views;
- Constructs supporting automatic consistency maintenance among user interface components;
- Constructs providing automatic incremental display updates;
- Strong support in the language for the creation of well-structured, maintainable programs.

The high-level structure (or *architecture*) of Clock programs is specified using an object-oriented style of specification language. The *ClockWorks* programming environment [67] is a graphical editor allowing the development, structured viewing, and modification of Clock architectures.

The components making up the architecture are programmed in a pure functional language in the style of Haskell [56]. Functions in Clock avoid clumsy continuation or monad-based syntax while maintaining a referential transparency property.

As with most pure approaches to introducing imperative features to functional programming, the execution of Clock programs consists of a set of purely functional computations, bound together by an outer shell in which imperative effects such as I/O may take place. In Clock, this outer shell is implicitly specified via the architecture, relieving the programmer of the task of specifying when the functional computations take place, and how to pass values between them. The Clock system takes care of the sequencing of computations, as guided by constraints introduced by the architecture description.

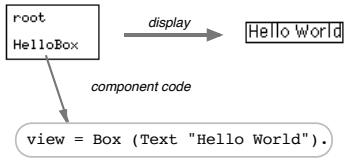


Figure 3.1: A complete Clock program displaying "Hello World". The program consists of a single component of class *"HelloBox"*; the component's text consists of a *view function* specifying the graphical display.

This chapter presents an overview of the features of the Clock language. The chapter is structured around a simple example, developed and extended throughout the chapter. Chapter 4 presents a more analytical view of Clock, describing the declarative properties of Clock, and discussing why Clock is a useful language for user interface development.

Clock has been implemented, and runs on Sun workstations under the X Window System [95]. In this chapter, all examples of Clock architectures are screen snapshots of the actual architecture developed within the ClockWorks environment, and examples of output generated by programs are screen snapshots of the running Clock program. The functional code given is the complete code required to create the applications.

3.1 A Minimal Clock Program

In Clock, programs are built as architectures of connected components. Architectures are specified in a graphical architecture editor within the *ClockWorks* programming environment [67]. Consider that we wish to write a program displaying a box containing the text *Hello World*. As shown in figure 3.1, this program would consist of a single component, whose program text is:

```
view = Box (Text "Hello World").
```

This code specifies that the view generated by the component is the value "`Box (Text "Hello World")`", which is displayed as a box surrounding the text *"Hello World"*. *view* is a function, with type definition:

```
view :: DisplayView.
```

where *DisplayView* is a predefined type representing possible graphical displays. Figure 3.1 shows how the single component looks in *ClockWorks*, and how the resulting view appears on the display.

The component in this example is called *root*, and is an instance of class *HelloBox*. The graphical output produced by the *HelloBox* component can easily be made more

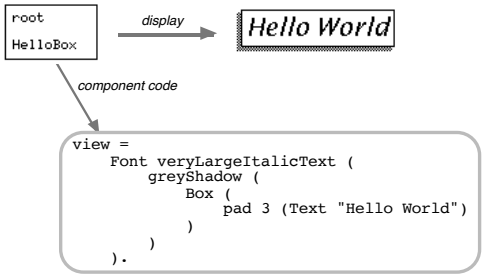


Figure 3.2: A complete Clock program displaying "Hello World" in a stylish manner.

aesthetic. Primitives in the Clock view language allow the specification of font, shadowing and positioning of display objects, leading to the new view function of figure 3.2. The *greyShadow* function is an example of one of the many predefined view functions provided in the Clock library; in particular,

```
greyShadow :: DisplayView -> DisplayView.
```

maps an arbitrary view to the same view with a grey shadow. Similarly,

```
pad :: Num -> DisplayView -> DisplayView.
```

takes an arbitrary view as parameter, and returns the same view with a padded border of the specified number of pixels. In addition to the library functions, programmers can also develop their own functions to manipulate views. For example, a neater way of coding the view from figure 3.2 would be to abstract the drawing of the box and shadow from the actual text used in the box. This is accomplished by creating a function *greetingBox* that makes a shadowed box around arbitrary text:

```
greetingBox :: String -> DisplayView.
greetingBox t = Font veryLargeText (
                    greyShadow (
                        Box (pad 3 (Text t)))).
```

Then, the view function can be specified as:

```
view = greetingBox "Hello World".
```

Views in Clock are elements of the *DisplayView* data type defined in the Clock library A partial definition of *DisplayView* is contained in figure 3.3 (the full definition is provided in chapter 4). *DisplayView* provides a simple set of primitives from which views can be built. The real power of the view language lies in the ease of specification of high-level view functions such as *greyShadow* and *greetingBox*.

```
% The basic display view data type.

data DisplayView =
    Views [DisplayView]                     % View grouping
  | Line Coord Coord                        % Line between given points
  | Arrow Coord Coord                       % Arrow between given points
  | At Coord Coord DisplayView              % Position view between given points
  | Box DisplayView                         % Box surrounding given view
  | Text String                             % Display text
  | NumText Num                             % Display number as text
  | CharText Char                           % Display character as text
  | BooleanText Bool                        % Display boolean as text
  | InstanceOf SubViewName SubViewId        % Display subview
  | Font FontVal DisplayView                % Set font for given view
  | Style StyleVal DisplayView.             % Set style for given view
```

Figure 3.3: The *DisplayView* data type forms the basis of view generation.

3.2 Trees of Components

In Clock, programs consist of trees of communicating components. Breaking programs up into components has several advantages – it assists in high-level organization of programs, and provides support for easily modifying and reusing code.

Figure 3.4 shows an architecture used to display greetings to "Stefan" and "Ulrich" side by side. The architecture to accomplish this display is split into three parts – the "stefan" component, which generates the *"Hello Stefan"* greeting; the "ulrich" component, which generates the greeting for Ulrich, and the "root" component, which combines the views of its two children to display the two greetings side by side.

The "GreetStefan" and "GreetUlrich" components each produce a view through the *greetingBox* function defined in section 3.1. This shows how easy it is in Clock to define interesting functions and use them in multiple contexts. The root component *Greetings* refers to its children "stefan" and "ulrich" when creating its own view. Components always refer to their children via a *subview name* rather than by the name of the component. This makes it easier to modify the architecture, since components do not need to know what they are attached to. The children components *GreetStefan* and *GreetUlrich* appear to the root component *Greetings* to be functions of type:

```
stefan :: String -> DisplayView.
ulrich :: String -> DisplayView.
```

Subview functions take a string parameter and deliver a display view. Section 3.3.2 shows how the string parameter can be used; for now, we simply use a null string.

Through this subview communication mechanism, the view of a child component is not directly displayed on the screen, but rather is passed up the tree to be used by its parent. The view finally generated by the root component is displayed on the screen.

The view function of the *Greetings* component uses the library function *beside*:

```
beside :: [DisplayView] -> DisplayView.
```

which lays out a list of views left to right.

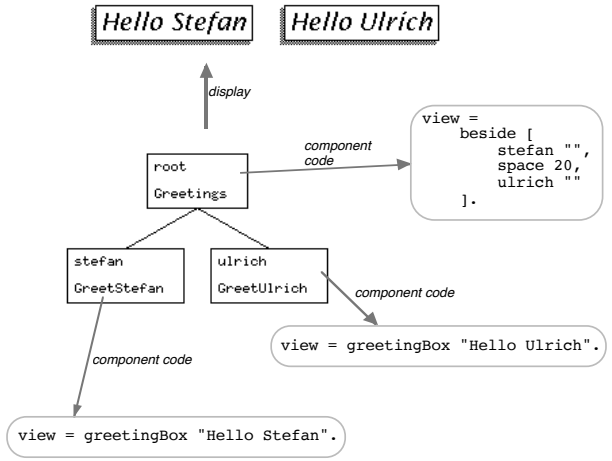


Figure 3.4: The greeting program is split into three architecture components.

3.3 Representing Persistent Data

The components we have seen so far are all called *event handler* components. These are components used to create an interactive display view. Because of their functional style, event handlers are stateless.

In order to represent state, a second form of component called *request handlers* is provided. Request handlers are a form of abstract data type – they have a hidden internal state, and a defined interface allowing requests about the state, and updates to the state. We shall first present an example of a request handler component, and then show how request handlers are used.

3.3.1 Example Request Handler

Request handlers are used to maintain and manipulate persistent state; thereby playing the role of data structures in Clock programs. Figure 3.5 shows an example of a request handler *Id*, implementing a simple string identifier. As shown in the architectural diagram, *Id* has two operations: *setMyId* is used to set the string identifier, while *myId* is used to query the value of the identifier.

The state of a request handler is modified by sending it an update event. For example, "`setMyId "Fred"`" is a value of type *UpdateEvent*. If "`setMyId "Fred"`" is sent to a component of class *Id*, then subsequent requests of "`myId`" will evaluate to the string `"Fred"`. Note that Clock programmers never need to create values of type *UpdateEvent*

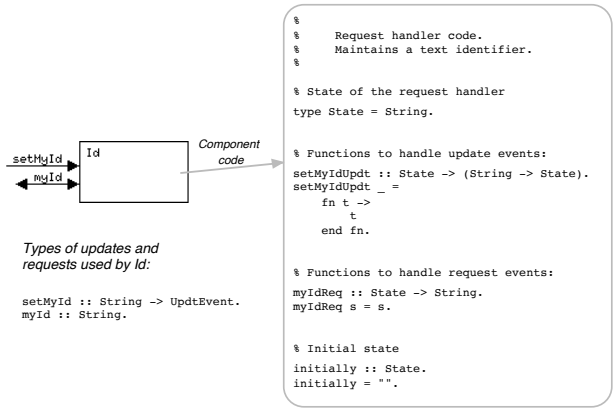


Figure 3.5: A Request Handler in Clock implementing a string identifier. The string value is set by sending the update *SetMyId* to the component; the string value is queried by sending the request *myId*.

explicitly; update events are created automatically through update functions such as *setMyId*.

As shown in figure 3.5, the code for a request handler requires: a type definition of the request handler's state; functions for each update and request the request handler is capable of handling, and an *initially* function specifying the components initial state.

The type *State* specifies the type of the component's state. The type of a request handler may be arbitrarily complex. In *Id*, the state is a *String*, since the identifier being represented is a string. The *initially* function returns a value of type *State*, specifying the initial state of the request handler; in this case, the initial value of *Id* is a null string, indicating there is no identifier.

The *update* functions in a request handler respond to update events sent to the request handler. An update function takes the current state as parameter, and delivers a new state as its result. Update functions are restricted to being purely functional.

For example, when an update "`setMyId "Fred"`" is sent to an instance of the *Id* component, the update function *setMyIdUpdt* is automatically applied, first to the current state, and then to the argument `"Fred"`. The result of the application becomes the new state of the request handler. In general, if an update function $u :: t_1 \rightarrow \ldots \rightarrow t_n \rightarrow updateEvent$ is listed in the interface of a request handler, then the request handler must have a function:

$$uUpdt :: State \rightarrow (t_1 \rightarrow \ldots \rightarrow t_n \rightarrow State).$$

to handle the update.

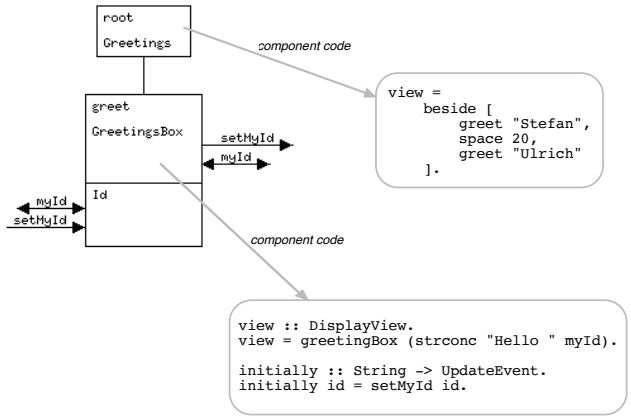


Figure 3.6: New version of greetings program of figure 3.4. The *GreetingsBox* component uses an *Id* component to represent the person to be greeted.

Similarly, if a request "`myId`" is sent to an instance of the *Id* component, the function *myIdReq* is applied to the current state, giving a string value as response to the request. In general, if a request function $r :: t_1 \rightarrow \ldots \rightarrow t_n$ is listed in the interface of a request handler, then the request handler must have a function:

$$rReq :: State \rightarrow (t_1 \rightarrow \ldots \rightarrow t_n).$$

to handle the request. Note that requests query the current state only, and cannot modify it.

Request handlers implement the imperative functionality of persistent data structures that can be queried and updated. The functions implementing request handlers are, however, purely functional, taking the current state as a parameter and returning values based on the state.

3.3.2 Using Request Handlers

As an example of how request handlers are used, consider that we wish to improve the implementation of the program of figure 3.4. In this implementation, we have two components *stefan* and *ulrich* implementing the greetings to Stefan and Ulrich respectively. These components have almost identical code, except that the name is different. If we make use of the *Id* request handler, we can replace the *stefan* and *ulrich* components with a single parameterized component.

Figure 3.6 shows this new program organization. The root event handler has a single subview *greet*. As usual, from the point of view of the root, *greet* has type:

```
greet :: String -> DisplayView.
```

The *greet* subview is implemented so that the string parameter is the name of the person to be greeted; i.e., "`greet "Stefan"`" evaluates to a greeting to Stefan, and "`greet "Ulrich"`" evaluates to a greeting to Ulrich. This allows us to recode the root component as:

```
view = beside [greet "Stefan", space 20, greet "Ulrich"].
```

Each use of *greet* creates a separate instance of the subview; these instances are identified by the string parameter to which *greet* is applied.

The *greet* subview is made up of a *GreetingsBox* event handler, and an *Id* request handler, as shown in figure 3.6. The annotations on the architecture diagram show that the *GreetingsBox* issues the update *setMyId*, and uses the request *myId*. These annotations are provided to make architectures visually easier to understand.

The code for the *GreetingsBox* event handler introduces the *intially* function, which is used to initialize a component when it is created. The function is typed:

```
initially :: String -> UpdateEvent.
```

When an instance of the *greet* subview is invoked with some string parameter s, if there is not yet any instance of s, an instance of *GreetingsBox* is automatically created, and the *initially* function of the new component is applied to s. In this example, s represents the name of the person to be greeted, so that separate instances of *GreetingsBox* are created for the strings `"Stefan"` and `"Ulrich"`. When applying the *initially* function of *GreetingsBox* to some name s, The update event "`setMyId` s" is passed to the *Id* component, setting the name of the person to be greeted.

The view function uses *Id* to obtain the name of the person to be greeted. In the code:

```
view = greetingBox (strconc "Hello " myId).
```

the request *myId* evaluates to the name of the person to be greeted. The full greeting is obtained by concatenating the string *"Hello "* to the name.

Requests appear to have the potential of violating referential transparency. For example, the expression "`streq myId myId`" should be a tautology, comparing two instances of the request *myId*. If we are unlucky, however, the two instances of *myId* could have different values, due to an update being performed during the evaluation of the expression. In fact, the execution semantics of Clock guarantee that requests maintain the same value throughout the execution of an expression. This is achieved by deferring potentially harmful updates until the expression has been completely evaluated. This guarantee means that all expressions that a Clock programmer can write are referentially transparent.

Since architectures may contain multiple instances of the same component, rules are required to specify how updates and requests are routed. These rules are quite simple.

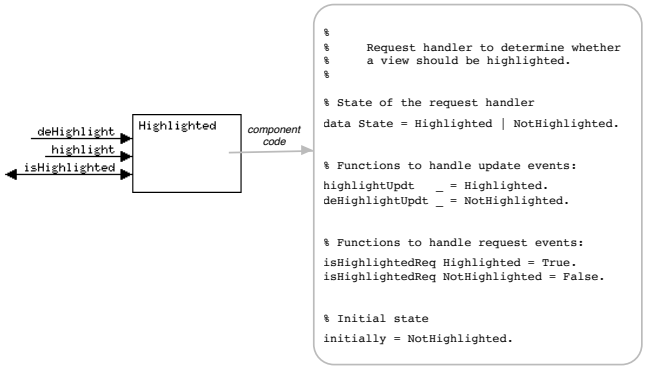


Figure 3.7: The *Highlighted* request handler and its implementation.

If an update or request cannot be handled by a *local* request handler (e.g., *Id* is *local* to *GreetingsBox*), then it is sent up the tree to the first component that is capable of handling it. Updates and requests always move up the architecture tree – never down or across the tree.

3.4 Input

So far, the programs we have created produce graphical output, but are not interactive. In order to create interactive programs, we need some way of expressing how user inputs are handled.

First, we introduce another useful request handler, called *Highlighted*. This request handler is used to determine whether a view is to be displayed in highlighted mode or not. As shown in figure 3.7, *Highlighted* takes two updates and one request:

```
highlight :: UpdateEvent.
deHighlight :: UpdateEvent.
isHighlighted :: Bool.
```

Imagine we wish to extend our greeting program so that if the user clicks on one of the greetings messages with the mouse, the message is drawn in inverted mode, and clicking again returns the message to normal mode. Figure 3.8 shows how this appears on the display, and how it is implemented in Clock.

The *greet* subview is extended to include a *Highlighted* request handler that represents whether the view produced by each instance of the subview should be inverted or not.

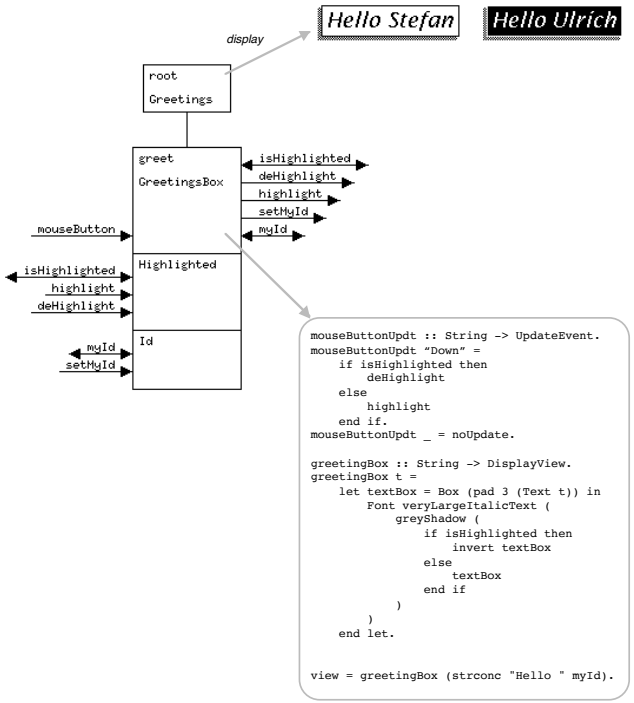


Figure 3.8: Clicking on one of the greetings inverts it; clicking again returns it to normal mode.

The architecture view of *GreetingsBox* shows that this event handler uses the updates and requests implemented by *Highlighted*.

The *GreetingsBox* event handler is tagged to show that it takes the *mouseButton* input. Mouse button inputs are created when the user clicks the mouse button. If the user clicks the mouse button on the display created by this component, a mouse button input is sent to the component.

In the code for *GreetingsBox*, the function

```
mouseButtonUpdt :: String -> UpdateEvent.
```

defines how the component responds to mouse input. The string parameter can be either *"Down"* or *"Up"*, corresponding the mouse button being depressed and released respectively. In this case, when the mouse button is depressed, the highlighted status of the display is toggled. This is achieved with the code:

```
mouseButtonUpdt "Down" =
    if isHighlighted then
        deHighlight
    else
        highlight
    end if.
```

This code states that when the mouse button is clicked down on this view, if the view is highlighted now, it should be de-highlighted, and vice versa. Recall that *highlight* and *deHighlight* are values of type *UpdateEvent*, which are sent to the *Highlighted* request handler. When the mouse button is released, nothing happens, as encoded by the special update event *noUpdate*:

The view of the component depends on the value of the *Highlighted* request handler. The *greetingBox* function is modified to invert the greeting if *highlighted* is true:

```
greetingBox :: String -> DisplayView.
greetingBox t =
    let textBox = Box (pad 3 (Text t)) in
        Font veryLargeItalicText (
            greyShadow (
                if isHighlighted then
                    invert textBox
                else
                    textBox
                end if
            )
        )
    end let.
```

```
% Keyboard inputs
key :: Char -> UpdateEvent.
arrowKey :: String -> UpdateEvent.      % Parm = {"Left", "Right", "Up", "Down"}
editKey :: String -> UpdateEvent.       % Parm = {"Tab", "Backspace", "Delete", "Escape", "Return"}
functionKey :: String -> UpdateEvent.   % Parm = {"1", "2", ...}

% Mouse button clicks
mouseButton :: String -> UpdateEvent.   % Parm = {"Up", "Down"}

% Mouse motion
enter :: UpdateEvent.                   % Mouse moves over display of this component
leave :: UpdateEvent.                   % Mouse moves away from display of this component
motion :: (Num, Num) -> UpdateEvent.    % Mouse moves to position (x,y)
relMotion :: (Num, Num) -> UpdateEvent. % Mouse moves (x,y) from last reported position
```

Figure 3.9: Predefined user inputs available in Clock.

In the *greetingBox* function, the *if*-expression evaluates to either the display view "`textBox`" or "`invert textBox`". The grey shadow is applied to whichever of these values is selected. The predefined function:

```
invert :: DisplayView -> DisplayView.
```

inverts its argument by swapping its background and foreground colours.

3.4.1 Views as Constraints

The *view* function of an event handler component specifies how, at any given time, the view of that component is to appear on the display. The view function of *GreetingsBox* differs from the view functions we have seen so far in that it does not always return the same value. When the user clicks on a greeting, the display presentation of the greeting is changed.

In Clock, programmers don't have to worry about *when* a view function is updated. The language guarantees that whenever a view function is out of date, it will be evaluated, and the new view will be posted on the display. View functions are therefore a specialized form of *constraint*, specifying the appearance of the display as a function of the current state of the program. As with other constraint-based languages, the programmer does not specify when the constraints must be evaluated; rather, the compiler automatically determines when updates are required. The current implementation of Clock performs data flow analysis to determine when views are potentially out of date, providing efficient incremental display updates.

3.4.2 Other Forms of Input

In addition to the mouse button input shown in the last example, Clock provides all of the input events in figure 3.9. Together, these inputs permit the implementation of

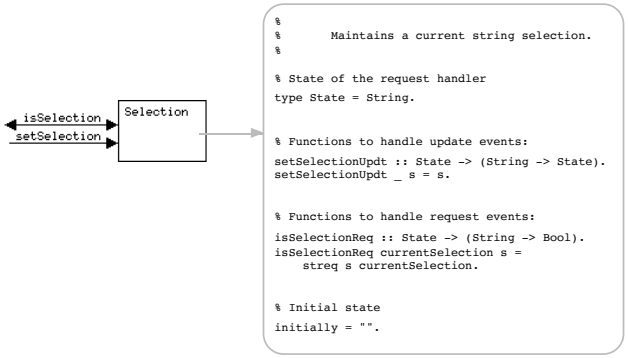


Figure 3.10: The *Selection* request handler.

all common interaction techniques found in direct manipulation user interfaces, such as menus, scroll bars, dialogue boxes, picking, dragging, and drawing.

3.5 Consistency Maintenance

Imagine that we wish to extend our example to provide a *radio button* functionality. Radio buttons have the property that only one button can be selected at any one time – if one button is clicked and highlighted, whatever button was highlighted before becomes de-highlighted.

Implementing radio buttons is an example of a consistency maintenance problem – the modification of one button's state requires the modification of any number of other buttons as well. Clock provides support for consistency maintenance through event handler *invariants*.

In this example, we require the new *Selection* request handler (figure 3.10). *Selection* implements a current string selection. The update:

```
setSelection :: String -> UpdateEvent.
```

sets the current selection; the request:

```
isSelection :: String -> Bool.
```

tests the given string to see whether it matches the selection.

Figure 3.11 shows an architecture implementing a radio button version of the grouping program. This program produces output as before, but permits only one greeting to

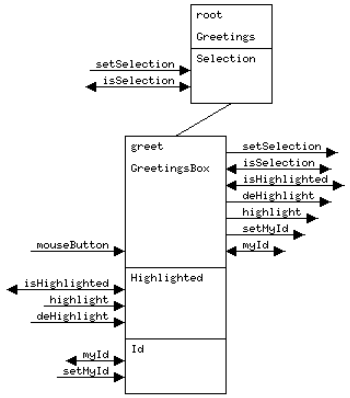


Figure 3.11: An architecture implementing radio buttons.

be highlighted at a time. Clicking on one greeting de-highlights any other highlighted greetings. The architecture diagram of figure 3.11 contains a great deal of detail, so in figure 3.12, we show a less detailed version of the architecture, in which local request handlers in the *greet* subview have been hidden.

As can be seen in the code of figure 3.12, whenever the user clicks on one of the greetings, the update "`setSelection myId`" is sent up the tree. This update sets the selected name in the *Selection* component to the identifier of the subview clicked.

Once the current selection changes, then the highlighting of the greetings may have to change. The subview whose identifier matches the selection must be highlighted, and the subviews that do not match the selection must be de-highlighted. In order to express this condition, we use an *invariant* function:

```
invariant :: UpdateEvent.
invariant =
    if isSelection myId then
        highlight
    else
        deHighlight
    end if.
```

The Clock semantics guarantee that whenever evaluating an invariant function would have an effect, the invariant will be evaluated. This means that whenever the selection is changed, the invariants of any subcomponents with incorrect highlighting will be evaluated. Invariants are therefore a form of constraint, used specifically for data consistency maintenance.

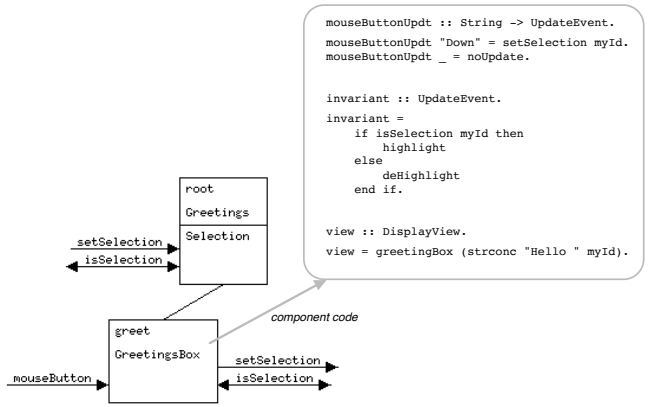


Figure 3.12: The architecture of figure 3.11, with information elided. The code for the *GreetingsBox* event handler shows the use of an *invariant* function for consistency maintenance.

An important restriction in Clock is that updates issued by invariants may only go to local request handlers. Philosophically, this restriction means that invariants are intended to express consistency conditions among local data. Practically, the restriction eliminates the danger of creating infinite sequences of invariants that trigger each other in cycles. Chapter 7 investigates termination conditions among invariant sequences.

3.6 Properties of Clock

As a language for specifying user interfaces, Clock provides many attractive features. The I/O language is simple yet powerful. High-level constraints allow easy specification of consistency maintenance and automatic, incremental view updates. Clock's architecture language allows high-level structuring of user interfaces. Request handlers provide easy access to persistent state. The language used to program components is functional, and therefore high-level and expressive; the functional programming is also in a direct style, as opposed to the continuation-based style found in many functional I/O systems.

Despite the inclusion of all of these features normally associated with imperative programming, Clock is purely declarative. Clock is *declarative in the small*, meaning that Clock functions are referentially transparent, and *declarative in the large*, meaning that the constraint system underlying Clock's view and invariant functions is also purely declarative. These properties are formally investigated in chapter 7.

3.6.1 Declarative in the Small

All expressions written by Clock programmers are guaranteed to be referentially transparent. This means that when a programmer writes a Clock function, he/she does not need to be aware of the order in which the function will be evaluated. The usual properties and transformation identities of functional programming (e.g., the *fold/unfold* transforms [11]) also hold of Clock functions.

Informally, referential transparency is present for the following reasons: functions in Clock are pure, except that they may contain requests and references to views generated by subviews. The values of requests and subviews change over the course of execution of a Clock program. However, when executing any individual Clock function (e.g., an invariant or a view function), all system state upon which the function depends is frozen. That is, the values of all requests and subviews are held constant during the execution of the function. This freezing of state implies that requests and subview references are referentially transparent for the duration of the execution of the function.

For example, the value of a view function may change from one invocation to the next. However, the values of the subviews and requests upon which the view function depends do not change during any single evaluation of the view function. The view function itself is not referentially transparent, but since the view function can only be invoked by the Clock system itself, this impurity is shielded from the functional code written by programmers.

3.6.2 Declarative in the Large

The execution of a Clock program consists of a set of purely functional computations. As we have seen, each functional computation can be the execution of an event, invariant or view function, or of an update or request function in a request handler. The Clock compiler is responsible for automatically determining when these purely functional computations should be triggered. We say that Clock provides declarative programming *in the large*, since Clock programmers never have to worry about when these computations are triggered, or in what order.

The formal properties of declarativeness in the large are discussed in chapter 7. Intuitively, the properties are as follows. Event functions handling user inputs are guaranteed to be executed in an order semantically equivalent to *fifo*. That is, whenever the order of handling user inputs matters, the input the user performed first must be processed first. Because of this fifo guarantee, the Clock programmer does not have to worry about when event functions are triggered.

Invariant functions are guaranteed to be triggered whenever their execution would make a change to local state. That is, if the local state has become inconsistent, it is guaranteed that the invariant function will be triggered to handle the inconsistency. Invariants may cause state changes that trigger other invariants; however, Clock's visibility rules guarantee that (i) it is impossible to write circular dependencies among invariants, and therefore (ii) any sequence of invariants triggering other invariants is

finite. Similarly, view functions are guaranteed to be triggered whenever a component's view is out of date. Clock programmers therefore need only be aware that when a state change occurs, the appropriate invariant and view computations will be triggered, and that the number of computations triggered is finite.

3.7 Conclusion

This chapter has introduced the features of the Clock language. We have seen that Clock provides flexible I/O based on a high-level view language, and flexible support for structuring through the Clock architecture language. Support for constraint-style consistency maintenance is provided through invariant functions.

The examples shown so far are, however, very simple, and do not give the flavour of how real applications are developed in Clock. Chapter 4 discusses how Clock is used to develop real interactive applications, and gives more extensive examples.

Chapter 4

Developing User Interfaces in Clock

Chapter 3 gave an overview of the Clock language and its properties. This chapter discusses the design of Clock, and how the language is used to implement graphical user interfaces. We show how Clock helps with the problems particular to user interface development, and how Clock's high-level declarative programming style aids in the evolutionary development of user interfaces. The chapter concludes with an evaluation of how Clock supports the problems of user interface development that were identified in chapter 2.

4.1 An Example User Interface

In order to show how Clock is used to develop substantial user interfaces, we return to the card file program first introduced in chapter 2. Figure 4.1 shows the simple name and address data base implemented via a card file metaphor. The current card displays the name, address and telephone number of the current person. A tab on the upper right corner of the card displays the first letter of the current person's surname.

Two mechanisms are provided for navigating among cards. On the card itself, arrow buttons allow the user to move to the previous or following card alphabetically. Advancing beyond the first or last card wraps around the card file.

Above the card, a row of letter buttons allows the user to move directly to the first card in a given letter. (We call these buttons a *slider*, analogously to a rolodex slider.) The use of a shadow gives these buttons a three-dimensional feel, where clicking on a button causes it to appear first depressed then released. The button corresponding to the current letter appears highlighted in reverse-video. If a button is selected for which there is no card, the next card alphabetically following that letter is selected.

The surname on the current card can be edited: if the user clicks on the surname, a cursor appears indicating a text insertion point. Pressing return or moving to another card commits the edit operation, causing the cursor to disappear. If the first letter

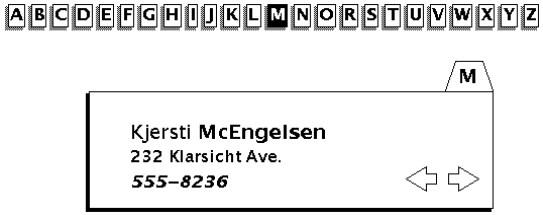


Figure 4.1: A card file name and address program implemented in Clock.

of the surname has been modified, the position of the card in the data base is modified to maintain alphabetical order, and the current letter tag and slider are updated accordingly.

Despite its apparent simplicity, this example shows that many details must be observed in creating a user interface, and indicates the numerous potential interactions between the components of a user interface.

4.2 The Clock Card File

Figure 4.2 shows a Clock architecture implementing the card file of figure 4.1. Clock architectures are structured as a tree of components. The tree structure supports a gradual progression from application to user interface, avoiding an artificially sharp split between the two. The root of the tree corresponds to the main application program, while the leaves of the tree implement the most basic interaction objects. In figure 4.2, the leaves of the tree correspond to the letter boxes making up the current letter selector, the editable text field used for the current surname, and the arrow buttons used to move between cards. The root of the tree represents the data base used to implement the card file.

Moving from the leaves up the tree, the leaf nodes are combined to form more abstract interaction objects. The *LetterBox* component is given a *Pushable* property; the pushable letter buttons are then combined to make the slider (*Slider*). The two arrow buttons are combined to make a card scroller (*Scroller*). The name and address data (in *NameAndAddress*) is combined with the current letter tag (in *Letter*) and the scroller to make up the current card (in *Card*).

A component may make use of zero or more instances of its children in making up its own display view. For example, the *Slider* component uses twenty-six pushable letter boxes in making up a slider.

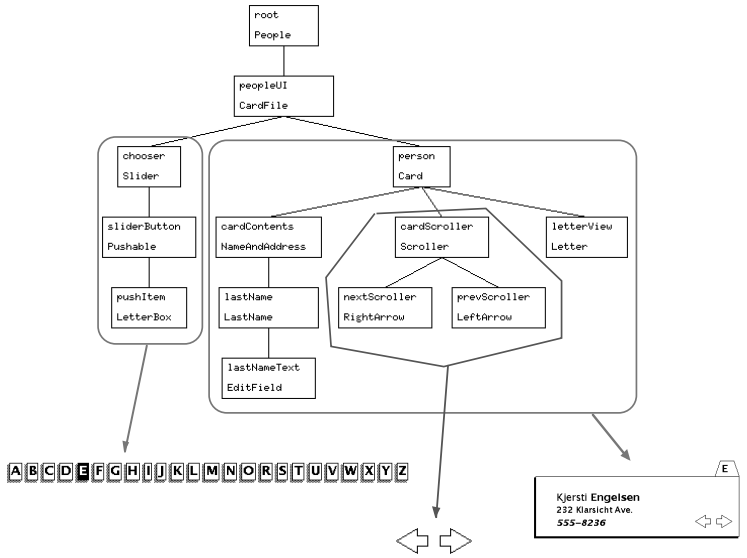


Figure 4.2: A Clock architecture for a card file name and address program.

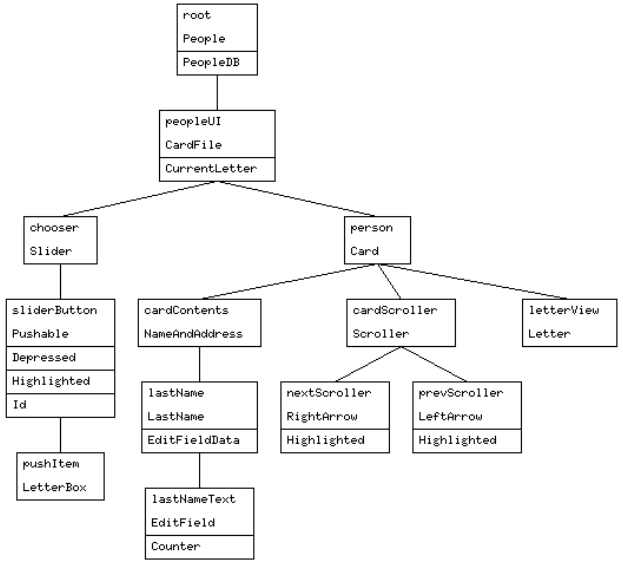


Figure 4.3: A Clock architecture with request handlers for a card file name and address program.

4.2.1 Structuring User Interfaces with Components

As was seen in chapter 3, architecture trees are composed of two kinds of components, *event handlers* and *request handlers*. Event handlers are stateless components responsible for maintaining display views and handling input. Request handlers make up the data structures of an architecture, each implementing an abstract data type usable by the architecture's event handlers. The card file provides interesting examples of complex interactions among components.

Figure 4.3 shows the full event handler/request handler breakdown of the cardfile architecture tree. The *PeopleDB* request handler represents the database of names and addresses. *CurrentLetter* represents the first letter of the current surname, and is used in the *Letter* and *Slider* event handlers.

As a more detailed example of the interaction between event handlers and request handlers, figure 4.4 shows the event handler *LeftArrow* with its request handler *Highlighted*. *LeftArrow* is responsible for maintaining a left-arrow graphic on the display, and for handling mouse input to the arrow. When the mouse button is depressed over the arrow, the arrow is redrawn in a highlighted (white-on-black) mode; when the

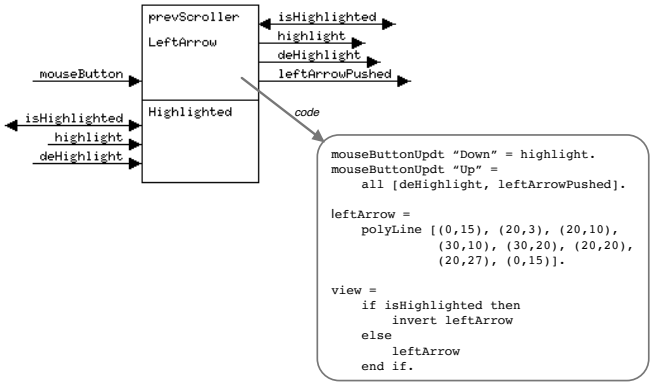


Figure 4.4: The *LeftArrow* event handler and its request handler *Highlighted*. The event handler maintains a display view and takes mouse input, while the request handler maintains the highlighted status of the display.

button is released, the arrow returns to normal black-on-white. The predefined function "`polyLine :: [(Num,Num)] -> DisplayView`" is used to specify the left arrow graphic.

The *Highlighted* request handler determines whether the arrow should be drawn in highlighted mode. In computing the view of *LeftArrow*, the request *isHighlighted* is used to determine whether the arrow is selected. The updates *highlight* and *deHighlight* are issued to *Highlighted* in response to mouse input.

Figure 4.5 shows the sequence of updates and requests resulting from clicking the mouse over the arrow graphic. The behaviour of the left-arrow interaction technique is that when the user clicks the mouse down on the picture of the arrow, the arrow is redrawn in inverted mode, giving visual feedback that the arrow has been clicked. When the mouse button is released, the arrow is redrawn in normal mode, and the update `leftArrowPushed` is sent up the tree.

As shown in the code of figure 4.4, clicking *down* on the arrow causes the update `highlight` to be sent to the *Highlighted* component. The view function is automatically invoked, and the inverted arrow is drawn. When the button is released, the two updates `deHighlight` and `leftArrowPushed` are sent. (The predefined function "`all :: [UpdateEvent] -> UpdateEvent`" allows multiple updates to be grouped into a single update.) Again, the view function is automatically updated to return the arrow to normal mode.

Clock programs consist of a set of pure functional computations. Each of these computations consists of the evaluation of a function defined in one of the request or event handlers – for example, a mouse input triggers the evaluation of an event handling

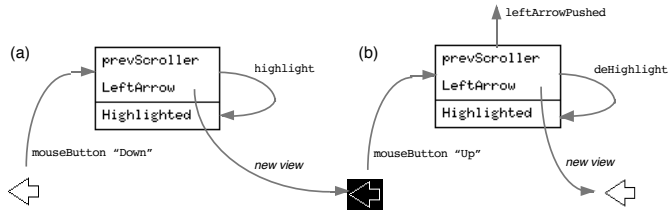


Figure 4.5: Sequence of events following clicking on the *LeftArrow* of figure 4.4. The highlighting of the arrow is used to give visual feedback that the click has been registered. The ultimate result of the click is that the event `leftArrowPushed` is sent up the architecture tree.

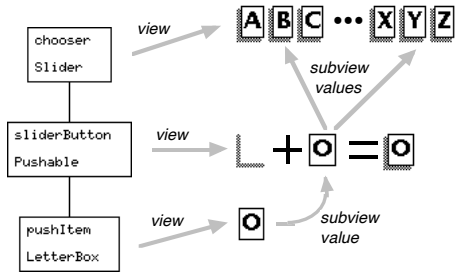


Figure 4.6: The value of each component is an interactive display view. Here, a push-button property is added to a letter box; a collection of pushable letter boxes is turned into a slider.

function, while an update may trigger the evaluation of a view function.

4.2.2 Views and Subviews

The value of event handler components is an interactive display view (or *view*) for short. In Clock, views are first class values: among other operations, they can be represented in variables, passed as parameters, and used to construct other views. A component can use the values of each of its children as subviews. Figure 4.6 shows an excerpt from the card file architecture, showing how views can be composed from subviews.

The value of a *LetterBox* component is simply a letter with a box around it. A *Pushable* component adds a push-button property to its subview, in this case the *LetterBox*. This behaviour adds a three-dimensional shadow effect to the letter box, and adds sensitivity to mouse clicks to implement a button behaviour.

The code for a simplified[1] version of *Pushable* is the following:

```
mouseButtonUpdt "Down" = depress.
mouseButtonUpdt "Up" = all [release, buttonPushed myId].

view =
    if isDepressed then
        pushItem ""
    else
        greyShadow (pushItem "")
    end if.
```

This version assumes that request handlers *Id* and *Depressed* are available, and that some subview called *pushItem* is to be given a pushable property. When the button is clicked *down*, the *pushItem* is depressed, resulting in its being drawn without a shadow; then the button is released, the *pushItem* is drawn with a shadow again. This gives the illusion of three-dimensional motion as the display object is "pushed" down and released. *Pushable* is an example of a component that is intended to be reusable in many architectures. The push-button property is generally applicable to any button used in a user interface, particularly since the *Pushable* component does not depend in any way on the view of its child.

The *Slider* component uses the values of twenty-six *Pushable LetterBox* components to create its own view. The twenty-six subviews are laid out horizontally, and a *radio button* behaviour is added, permitting exactly one of these subviews to be highlighted at any one time.

The view function in *Slider* is:

```
alphabet =
    ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L",
     "M", "N", "O", "R", "S", "T", "U", "V", "W", "X", "Y" ,"Z"].

buttons = map sliderButton alphabet.

view =
    Font normalBoldText (
        beside (insertBetween buttons (space 3))
    ).
```

Recall that to *Slider*, the subview *sliderButton* appears to be a function "`sliderButton :: String -> DisplayView`". By mapping *sliderButton* over the alphabet, the list of buttons is created. The predefined function "`insertBetween :: [DisplayView] -> Num -> [DisplayView]`" is used to space the buttons apart; the function "`beside :: [DisplayView] -> DisplayView`" lays out the buttons left to right.

[1] The real *Pushable* component also takes care of highlighting.

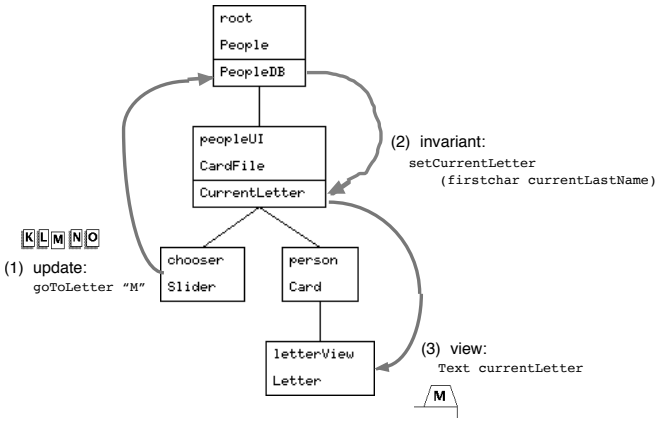


Figure 4.7: Communication between components via updates and requests. Clicking the slider letter "M" initiates a sequence of updates and requests.

4.2.3 Consistency Constraints

In implementing the card file of figure 4.1, a number of consistency conditions must be observed. The main condition is: the highlighted letter in the slider, the letter in the card tab, and the first letter of the surname in the card must all be the same. This letter may change from any of: scrolling over a letter boundary; clicking a letter button, or editing the surname on the card.

In the Clock card file, these consistency conditions are implemented automatically through the constraints provided by view and invariant functions. In general, view functions are sufficient to implement almost all consistency conditions; most components do not require an invariant function.

Figure 4.7 shows the interaction of view and invariant functions to implement consistency conditions. The figure shows the sequence of function evaluations resulting from clicking the *"M"* slider button. As we saw in section 4.2.2, clicking a letter button results in the update "`buttonPushed myId`" being sent up the tree. This button-pushed update is handled by an event function in *Slider*:

```
buttonPushedUpdt :: String -> UpdateEvent.
buttonPushedUpdt letter = goToLetter letter.
```

This function specifies that clicking the *"M"* slider button causes the update *goToLetter "M"* to be sent up the tree. The *goToLetter* update function has type "`goToLetter :: String -> UpdateEvent`".

The Clock system automatically sends the update event up the tree to the first event or request handler that is able to accept it. In this case, the *PeopleDB* request handler accepts the update, setting the current person to the first person whose surname begins with the letter *"M"*.

The modification of the state of *PeopleDB* causes an inconsistency between the state of *PeopleDB* and the request handler *CurrentLetter*, which still refers to the old letter. The *CardFile* event handler contains an *invariant* condition that specifies how the state of *CurrentLetter* is to be kept consistent with that of *PeopleDB*:

```
invariant = setCurrentLetter (firstchar currentLastName).
```

This invariant takes the current last name from *PeopleDB*, and uses its first character to set the currentLetter. The invariant uses the request "`currentLastName :: String`" which is handled by *PeopleDB*.

Finally, the modification to *CurrentLetter* causes the view of the letter tag on the card display to be out of date. The view of the letter tag is specified in *Letter* with the function:

```
view = pad 3 (Text currentLetter).
```

specifying that the current letter (obtained via a request to *CurrentLetter*) is to be displayed with a border of three pixels around it.

This example shows how clicking a letter button changes the current card in *PeopleDB*, resulting in a change in the *CurrentLetter* request handler. The components which depend on the current letter (*Slider* and *Letter*) can base their view functions on *CurrentLetter*, allowing their views to be automatically updated whenever the current letter changes.

In this example, we see that a simple action (the user clicking the mouse) can lead to a great deal of activity in the Clock architecture – the original mouse click led to the `buttonPushed` update, which in turn triggered the *goToLetter* update, which triggered an invariant function, which in turn triggered a number of view functions. This sequence of updates triggering more updates, invariants and views is similar to the chains of dependent constraints of languages such as Garnet. Clock differs from Garnet, however, in that every sequence of triggered constraints is guaranteed to be finite. The basic argument for this finiteness is simple – updates travel up the tree, and therefore sequences of updates must eventually terminate at (or before) the root component. Invariants can only make local updates based on requests made up the tree. Therefore, invariants can only trigger invariants lower in the the tree. Since trees have finite depth, sequences of invariants triggering invariants must also eventually terminate. These arguments are formalized in chapter 7.

4.3 Modifying Architectures

A major design goal in Clock is to support the *iterative design* of user interfaces. The principle behind iterative design is programmers cannot simply design a user interface and then implement it. User interfaces must be refined based on the results of user evaluation. Usually, several iterations of testing and modification are required to create a satisfactory user interface. Hence, it should be easy to quickly modify user interfaces in order to rapidly test new designs, or react to problems users encounter.

One major problem with iterative refinement is that as programs are substantially modified from their initial design, the overall structure of the program can degrade. Programs that have been developed using this experimental approach can easily become difficult to understand, maintain, and modify.

Clock provides four features that support the iterative refinement of user interfaces. These features also help to maintain structure despite modification:

Easy Reuse: Clock comes with a library of predefined components. Often, many of the components required in a change can be drawn straight from the library. Using library components speeds up modifications by reducing coding time, and helps in maintaining structure, since predefined components have been designed to be easily and cleanly connected.

Untargetted Updates/Requests: Updates and requests are *values* that are sent up the architecture tree, and handled by the first component that knows how to deal with them. This means that an event handler issuing an update or request does not have to be aware of the structure of the tree above it. If the structure of the tree above an event handler is modified, the event handler itself does not have to be modified, as long its updates and requests continue to be serviced.

The fact that updates and requests are untargetted also aids in component reuse. Since components do not have knowledge of the context in which they are used, it is easy to connect existing components into new architectures.

In the constraint-based approaches of languages like Garnet, constraints are explicitly tied to objects. To achieve component reuse, Garnet must introduce *pointer variables*, permitting explicit indirection. The need to set and reset these pointer variables dynamically is one of the reasons why Garnet-style constraint systems require side-effects.

Named Subviews: In Clock, the names of subviews are tied to the parent class, not to the child. This means that an event handler does not need to have any knowledge about how its subviews are computed. Subview components can therefore be replaced or modified without any modification to their parents whatsoever.

Restricted Visibility: In Clock, updates and requests travel *up* the tree only. This means that it is only possible for data dependencies to exist between a component, its children, and its parents, since it is not possible to express communication laterally across the tree. These visibility controls restrict the number

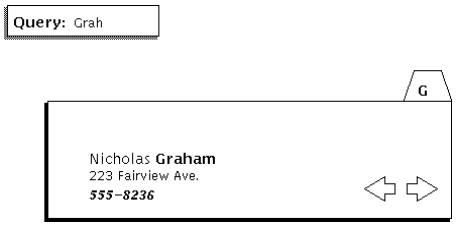


Figure 4.8: Modified card file program, where cards are selected by typing in a fragment of the last name.

of components that may be affected when a component is modified. These restrictions, therefore, help in controlling the effects of change, and in maintaining structure when change takes place.

Figure 4.8 shows an example of a modification to the card file program of figure 4.1. In the new version, rather than using a rolodex-style slider to select cards, we use a text field allowing the user to type a leading string of the desired last name. As shown in the figure, the string *"Grah"* is sufficient to locate the card for *"Graham"*. Such a text-based interface might be suitable for locating data in a large data base of cards.

Figure 4.9 shows the modifications that have to be made to accomplish this change. The *chooser* subview of the *CardFile* component is replaced. Instead of the old *Slider* component, we now have a *NameChooser* component implementing the query box shown in figure 4.8. When a new name is selected (e.g., the leading string *"Grah"*), the update "`goToName`" is issued to move to that name; the *PeopleDB* request handler (attached to the *People* component) was modified to handle this request.

The *NameChooser* component is based on the *EditField* component taken from the library. Note that *EditField* is also used to implement editing of the data on cards. *NameChooser* is new, and had to be programmed.

Despite the fact that the card selection mechanism was completely changed, no other components had to be modified. From the point of view of the *CardFile* component, the *chooser* subview is still responsible for choosing which card to view; *CardFile* has no knowledge of how the the selection is accomplished, and is therefore not affected when the mechanism is changed. Changing which card is currently displayed is handled automatically by the view functions of the components in the *person* subtree; therefore, these components are not affected by the change in the *chooser* subview.

This example demonstrates how well Clock localizes changes. Because of the flexible support for architecture modification, the changes were easy to make. Because of the restricted visibility of Clock architectures, the changes are guaranteed to be localized to a small number of components. The localization of change also implies that the structure of the architecture as a whole is minimally affected by the change of the selection mechanism. This minimal impact on the architecture helps in maintaining

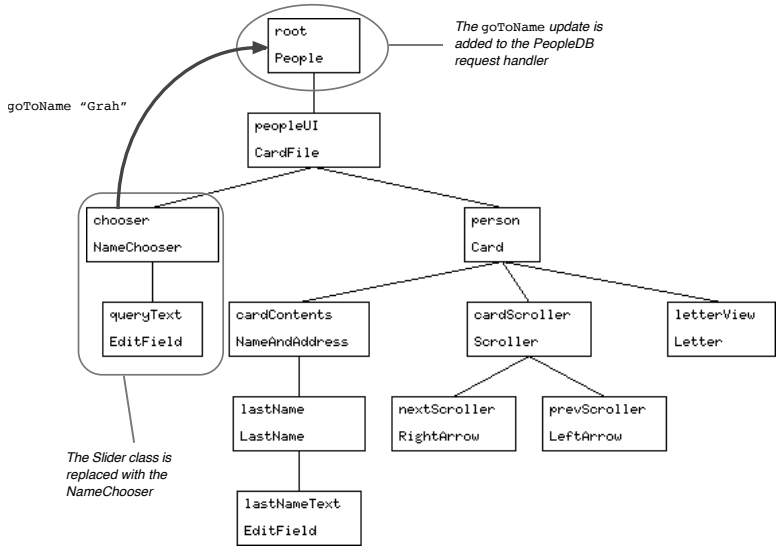


Figure 4.9: The architecture for the modified card file program. The components that had to be modified are shown.

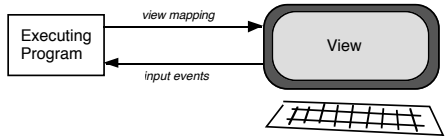


Figure 4.10: The Relational View Model.

clean structure throughout iterative design.

4.4 The Clock I/O Model

Clock's I/O system is based on the declarative *Relational View Model* of input and output [43]. Under the relational view model, there is considered to be a two-way mapping between an executing program and its user interface. A *view* mapping specifies the appearance of the display as a function of the data state of the program. An event-based mapping in the other direction allows manipulations of the user interface to be reflected in the state of the executing program. The two mappings together can be considered to be a relation, since they guarantee that the state of the user interface and the program are always consistent.

In Clock, the relational view model is implemented via the view and event functions provided in event handlers. The view function is a mapping from the state of the program (as obtained via requests) onto a display (an element of the type *DisplayView*.) Event functions process events generated by the user (such as keyboard input, mouse clicking or mouse motion events), and modify the state of the program via updates. Thus, the view and event functions together ensure that the program state and display state are always consistent.

Clock's I/O model is declarative, in that a pure function maps a snapshot of the system state onto a display value. The Clock system is then responsible for posting the view on a workstation display. The view function is automatically evaluated as often as necessary to maintain an up to date display.

The relational view model differs from constraint-based approaches such as ALV [48] or Garnet [70]. These approaches use the *retained object* model of output, where an object must be explicitly created in memory for each display primitive. Constraints are used to maintain consistency between application and display objects, in effect implementing a relation between the two. As the display is modified, however, the programmer must explicitly create and destroy view objects to keep the display up to date. This object management is performed via side-effects in the consistency constraints. Unlike the relational view model, therefore, the programmer must be aware of what has been on the display in the past, and of when the side-effects in the constraints will be executed.

```
% The basic display view data type.
data DisplayView =
    Views [DisplayView]
  | Line Coord Coord
  | Arrow Coord Coord
  | At Coord Coord DisplayView
  | Box DisplayView
  | Text String
  | NumText Num
  | CharText Char
  | BooleanText Bool
  | InstanceOf SubViewName SubViewId
  | Font FontVal DisplayView
  | Style StyleVal DisplayView.

type SubViewName = String.
type SubViewId = String.

type Char = Num.
type FontVal = String.
```

```
% Definition of coordinates as used in the At,
% Line and Arrow constructs.
type Coord = (Ordinate,Ordinate).
type Offset = Num.
type OrdinateLabel = Num.
type OrdinateLocation = Num.

data Ordinate =
    XBaseOffset OrdinateLabel Offset
  | YBaseOffset OrdinateLabel Offset
  | Left Offset  | Bottom Offset
  | Right Offset | Top Offset
  | XSomewhere | YSomewhere.

%      The available grahical styles
data StyleVal =
    Invisible | Solid | Filled | Dashed
  | Dotted | Bold | Grey | White
  | Elipse | BoldElipse | ThickElipse
  | Thick | Inverted | LightGrey
  | NoBorderGrey | NoBorderLightGrey.
```

Figure 4.11: The *DisplayView* data type forms the basis of view generation.

4.4.1 The Clock View Language

The Clock view language is based on the functional *Graphical View Language* [18]. Views must be elements of the data type *DisplayView*, as defined in figure 4.11. This view language is based on a very simple set of primitives. By taking advantage of the functional abstraction mechanisms of the Clock language, powerful view functions are built from the *DisplayView* primitives. Examples we have seen include *beside*, which lays out a set of views left to right, and *polyLine*, which generates a set of lines to connect the given coordinates.

Based on the predefined view functions, view specification is simple and high-level. It is rarely necessary to include explicit coordinates to specify the size or position of views. The full expressive power of the underlying view language is, however, always available to the programmer.

4.5 Conclusion

In chapter 2, we identified a set of problems particular to user interface development. We conclude this chapter with an examination of how Clock addresses each of these problems.

Iterative Refinement: Iterative refinement requires the ability to rapidly prototype and modify user interfaces. Clock aids rapid prototyping and modification on three levels. Firstly, the Clock architecture language allows replacement of interaction techniques (represented by architecture subtrees) with minimal modification to the rest of the tree. This plug-replacement also supports component reuse. Reuse not only frees the programmer from repetitively recoding the same

behaviours, but also provides the architecture programmer with a vocabulary of predefined components, leading to better structured architectures.

Secondly, within event handlers, invariant and view functions are automatically invoked as necessary to maintain up-to-date internal state and display views. This frees the programmer from having to determine when these updates are necessary, and from having to identify all the complex interrelations between user interface components.

Finally, the use of functional style makes component programming very high level. Our experience has shown that pseudo-code descriptions of event handler functions map almost directly into functional programs. Particularly within view functions, the fact that views are first class values and the use of higher order functions combine to give compact and elegant descriptions.

Direct Manipulation: Direct manipulation style interfaces are hard to program because user inputs may come in an arbitrary order, directed to arbitrary parts of the user interface. In Clock, inputs to a display view are automatically directed to the event handler component that produced the display. Each component need only know how to handle the inputs directed to it, without needing to be aware of inputs directed to other components. When components have interdependencies, invariant functions automatically reflect changes in state caused by inputs elsewhere in the system.

Semantic Feedback: Semantic feedback in a user interface component is used to obtain application data within an interaction technique. In Clock, semantic feedback is implemented through the request mechanism. Requests may directly access application data arbitrarily far up the architecture tree. Since requests are not explicitly targetted to any particular request handler, they do not reveal the structure of the application, thereby preserving reasonable separation of concerns between the user interface and application. A component making requests for application data can be reused in any architecture tree, as long as the request is served somewhere in the tree.

Communication among User Interface Components: In Clock, even when components are interdependent, they need never communicate with each other directly. The invariant mechanism allows a component to be automatically updated following the modification of another component upon which it depends. This means that components can be defined independently of another, easing clarity in architectures, and aiding component reuse.

User Interface Consistency: Clock contains a class concept allowing components to be defined once, and reused over an architecture. This mechanism allows the capturing of interaction techniques such as buttons and menus, and of behaviours, such as selection and searching. Reuse naturally guarantees consistency, helping improve the quality of the user interface.

Concurrency: The current implementation of Clock does not support concurrency, but the semantics of the language permit concurrent implementations. One possible approach to concurrency would be the component as process model, where one event handler processes an input, while another concurrently processes the next input. The semantics guarantee that as long as these inputs do not conflict with one another, a concurrent implementation is valid.

The semantics also could permit multi-user implementations. One possibility would be to split the architecture over a client-server network. The root application and its data would be placed on the server, while each user would be a subview instance of the client, communicating updates and requests over the network. Work is ongoing investigating this approach.

This concludes our description of the Clock language. We have stated that Clock helps solve the problem of declarative programming of user interfaces by combining a flexible I/O system with high-level support for structuring and consistency maintenance, and yet is purely declarative. This chapter has shown how Clock is used to build user interfaces. It remains to be demonstrated, however, that Clock actually is purely declarative – i.e., that Clock functions are referentially transparent, and that the programmer does not need to be aware of the order of evaluation of Clock constraints.

To reason about Clock's features, we introduce in chapter 5 the Temporal Constraint Functional Programming framework (TCFP), upon which Clock is based. In chapter 6, we define the formal semantics of Clock based on TCFP. In chapter 7, we use TCFP's proof mechanisms to state and prove the declarative properties of Clock.

Chapter 5

Temporal Constraint Functional Programming

In the description of Clock in chapter 4, we stated that Clock satisfies two basic properties of declarative user interface programming: referential transparency, and freedom from constraint cycles. To substantiate these claims, we must formally specify the semantics of Clock, and based on the semantics, we must prove that the properties hold. With this goal in mind, chapter 6 defines the semantics of Clock, while chapter 7 provides proofs of the declarative properties.

This chapter presents *Temporal Constraint Functional Programming* (or TCFP), a novel framework for defining the semantics of I/O, non-determinism and concurrency in extended functional languages. Because of these facilities, TCFP is an appropriate formalism for defining the semantics of the Clock language.

TCFP specifications of language semantics are split into two levels: an extended λ-calculus (called the *Interaction λ-Calculus*) is used to define the core of the functional language, while constraints in a temporal logic (the *Interaction Logic*) are used to specify the meaning of constructs supporting interaction and concurrency.

Because of its basis in the temporal interaction logic, TCFP provides strong support for reasoning about languages. If a language is defined using TCFP, then proving a property of the language consists of first specifying the property as a formula of interaction logic, and proving the formula to be a consequence of the language's semantics. Proofs are therefore based on standard techniques of logical reasoning. This approach is followed in chapter 7 to prove the declarative properties of the Clock language.

In this chapter, we first provide an informal overview of the TCFP framework. While this overview is intended to provide sufficient information to understand chapters 6 and 7, the fully formal definition of TCFP is also provided in appendices A, B and C.

Following this informal introduction, we give some examples of how TCFP is used. These examples show how TCFP can be used to model processes, synchronous and asynchronous interprocess communication, variables, and oracles. These facilities are defined as predicates in interaction logic, and collectively form the building blocks from which the semantics of complex languages such as Clock can be built.

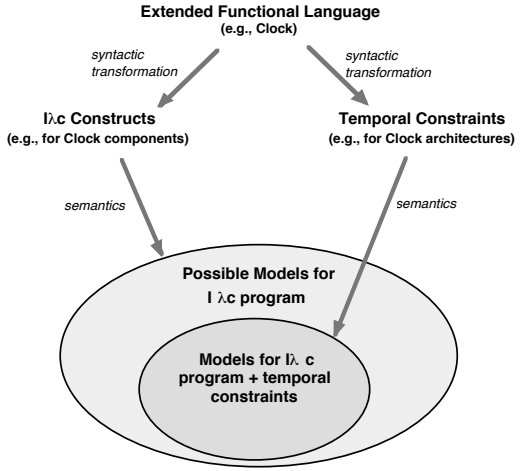


Figure 5.1: Defining extended functional languages using TCFP: the language is defined as syntactic sugar for TCFP constructs. The semantics of TCFP then give meaning to the constructs of the language. For example, in the definition of Clock (chapter 6), the functions defining Clock components are mapped to $I\lambda c$ constructs, while the architecture structure is mapped to interaction logic constraints. Combined, these give the meaning of Clock programs.

5.1 Informal Introduction to TCFP

The TCFP framework is a combination of two formalisms: $I\lambda c$, which is used to define extended functional constructs, and *interaction logic*, which is used to specify structural and temporal aspects of extended functional languages. As shown in figure 5.1, extended functional languages such as Clock can be defined by considering them to be syntactic sugar for TCFP constructs. As we shall see in chapter 6, the functions used to define Clock components are mapped into $I\lambda c$ functions, while the architecture specification of a Clock program is mapped onto constraints in interaction logic.

The semantics of an extended functional language are therefore given through the semantics of TCFP itself. As shown in figure 5.1, TCFP's semantics are based on a *generate and restrict* approach. $I\lambda c$ contains non-deterministic constructs, meaning that any given $I\lambda c$ program can have multiple possible executions. Any observable execution of an $I\lambda c$ program is called a *model* of the program. We therefore say that $I\lambda c$ programs *generate* a set of possible models. Temporal constraints serve to rule out execution cases we consider to be incorrect, therefore leading to a *restricted* set of models.

As an analogy illustrating how the *generate and restrict* approach works, consider the rules governing when a person may drive a car in Canada's province of Ontario. In general, anyone can drive at any time, but with a series of exceptions: people must be at least 16 years old with a valid driver's license; new drivers may not drive at night, or on highways; people with vision problems must wear their glasses to drive.

The *generation* component of this specification states the overly general case that anybody can drive at any time. The *restriction* component narrows this set of possibilities by imposing constraints on age, time of day, etc. As human beings, we don't have great difficulty understanding this form of definition – we understand that driving is generally permitted, as long as the myriad of constraints is satisfied.

We now give a simple example of a system specified using TCFP, after which a more detailed introduction to the formalism is given.

5.1.1 A TCFP Example

The core of TCFP is the *Interaction λ-calculus*, or $I\lambda c$ for short. As is shown in appendix A, $I\lambda c$ is based on an extended version of the untyped λ-calculus. Through the use of syntactic sugar, $I\lambda c$ can be extended to contain the constructs found in most modern functional languages, such as higher-order functions, lambda abstractions, pattern matching, equational definitions, let-bindings and case selection; the resulting syntax is similar to that of Haskell [56]. For example, an $I\lambda c$ program to reverse a list is:

```
append [] b = [b].
append (a:as) b = a : (append as b).

reverse [] = [].
reverse (a:as) = append (reverse as) a.
```

For simplicity, $I\lambda c$ is untyped, so that $I\lambda c$ expressions have values in some domain that we call *value*. Practical languages built from $I\lambda c$ would normally introduce some form of typing system.

$I\lambda c$ extends normal functional programming with constructs to express input and output. For example, the expression:

```
read "stdin"
```

reads from a communication port called *"stdin"*; the value of the expression is the value read. Similarly, the expression:

```
write "stdout" "Hello"
```

writes the value *"Hello"* to the communication port *"stdout"*, and the value of the expression is the value written (here, the string *"Hello"*). $I\lambda c$ assigns no particular

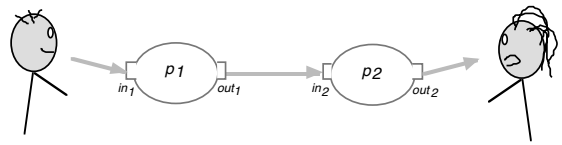


Figure 5.2: Two communicating processes. Process p_1 takes inputs from communication port in_1, and writes the values to communication port out_1. Process p_2 reads from in_2, and writes the values to out_2. Such a structure might be used to allow two people on different machines to communicate.

meaning to these communication ports; as we shall see in the next section, constraints in interaction logic are used to specify the behaviour of communication ports, and how they are connected.

$I\lambda c$ does not specify any order in which I/O must take place. In the expression:

```
(write "stdout" "Hello", write "stdout" "world")
```

the output to *"stdout"* may be `"hello world"` or `"world hello"`.

Because the $I\lambda c$ *read* function does not always return the same value, $I\lambda c$ expressions are not guaranteed to be referentially transparent (see theorem A.3.1).

Constraints in interaction logic can be used to rule out possible executions of the program that we do not consider to be correct. For example, the constraint:

$$out(\texttt{"stdout"}, \texttt{"world"}) \supset \blacklozenge\, out(\texttt{"stdout"}, \texttt{"Hello"}).$$

states that if the value `"world"` is written to the communication port `"stdout"` *now*, then *sometime in the past* (written "$\blacklozenge$"), the value `"Hello"` must have been written to `"stdout"`. This constraint rules out the undesirable case of writing `"world hello"`, since the only way `"world"` can be written is if `"Hello"` is written first. The predicates *in* and *out* are predefined in interaction logic to allow constraints to be placed on input and output activity performed by $I\lambda c$ programs.

Interaction logic is a temporal logic, including all the facilities of classical, first order predicate logic, as well as a set of temporal operators (such as "$\blacklozenge$") to allow temporal reasoning. All of the traditional axioms and proof rules of classical logic continue to hold in interaction logic, so that traditional reasoning techniques continue to be applicable.

An Example: Modelling Interprocess Communication with TCFP

TCFP can be used to model complex systems such as processes and interprocess communication. Consider, for example, that we wish to model two processes (figure 5.2). The first reads inputs from communication port in_1, and writes the same values to

communication port out_1. The second process reads the values that the first process sent, and writes them to its own output port. Such a configuration might be used to allow two people to communicate over a network.

In $I\lambda c$, we define the processes as:

```
process inCP outCP =
    (write outCP (read inCP)) : (process inCP outCP).

eval (process "in1" "out1", process "in2" "out2").
```

That is, two invocations of the *process* function are used to model the processes. Since $I\lambda c$ has no defined control flow, it is possible for the processes to be evaluated sequentially, or in parallel. Each process repeatedly reads from its "in" communication port, and writes the same value to its "out" communication port.

This program is non-deterministic, since it is not specified what values will be read from the input communication ports. In particular, there is nothing in the $I\lambda c$ code to specify that the output of process p_1 becomes the input of process p_2. The semantics of $I\lambda c$ simply specify that whenever a *read* occurs, some non-deterministically selected value is read. We will use interaction logic to constrain this non-deterministic selection so that the output from process p_1 always becomes the input from process p_2.

First, we can express that whatever p_2 reads must have been output sometime earlier by p_1. This is stated as:

$$\forall v : value, i : uid \,.\, in(\texttt{"in2"}, v, i) \supset \blacklozenge\, out(\texttt{"out1"}, v, i).$$

This constraint states that for every value v read over communication port in_2, the same value must have been written sometime in the past over communication port out_1. The operator "$\blacklozenge$" (read *"was sometime"*) indicates that the predicate "$out(\texttt{"out1"}, v, i)$" must have been true sometime in the past. In order to distinguish between different I/O events involving the same value, TCFP assigns a unique identifier to each I/O event. Therefore, "$in(\texttt{"in1"}, v, i)$" can be read as "the value v is read now over communication port in_1, and this read operation has unique identifier i."

This constraint specifies that any values read by p_2 must have been written earlier by p_1. However, it does not state that all values written by p_1 must eventually be read by p_2. This form of communication could describe low-level network communication, where it is not guaranteed that every value sent over the network will arrive at the other end. If we wish to guarantee that all values from p_1 are eventually read by p_2, we can add the following constraint:

$$\forall v : value, i : uid \,.\, out(\texttt{"out1"}, v, i) \supset \lozenge\, in(\texttt{"in2"}, v, i).$$

This constraint specifies that if a value v (with unique identifier i) is written over communication port out_1, then sometime in the future (written "$\lozenge$"), the same value (with the same unique identifier) must be read from communication port in_2.

Finally, while these constraints specify that the output from p_1 must eventually become the input to p_2, nothing is said about the *order* in which the values must be read. This

form of communication could describe a lossless networking protocol, where all values are guaranteed to arrive at the destination, but where the order may be jumbled. If we wish to specify a *fifo* ordering, we can use the constraint:

$$\forall v_1, v_2 : value, i_1, i_2 : uid \,.\\ (out(\texttt{"out1"}, v_1, i_1) \lhd out(\texttt{"out1"}, v_2, i_2)) \\ \supset (in(\texttt{"in2"}, v_1, i_1) \lhd in(\texttt{"in2"}, v_2, i_2)).$$

This constraint specifies that if the output of v_1 *precedes* (written $\lhd$) the output of v_2, then the input of v_1 must also precede the input of v_2, thereby preserving the ordering of values sent.

This example has shown how a single $I\lambda c$ program can be given different properties by applying different temporal constraints. The example also shows that the integration of $I\lambda c$ and integration logic is simple, in that interaction logic can be used to constrain the execution of $I\lambda c$ programs by restricting their I/O activity.

It should be emphasized, however, that TCFP is not a programming language in itself. TCFP is to powerful to be implemented directly. The real use for TCFP is as a means for expressing the semantics of programming languages, as shown in figure 5.1. Section 5.3 shows numerous examples of how small, extended languages can be specified with TCFP. First, however, the next section gives a more detailed overview of TCFP.

5.2 Syntax and Informal Semantics of TCFP

The last sections presented an example giving the flavour of TCFP, and showing how TCFP can be used to model impure extensions to functional languages. The TCFP framework itself is fully described in appendices A, B, and C. This section gives a less formal introduction to TCFP, summarizing the basic properties of the framework.

5.2.1 Interaction Lambda-Calculus

The functional component of the TCFP framework is based on a language called the *Interaction Lambda Calculus*, or $I\lambda c$ for short. As described in appendix A, $I\lambda c$ consists of a small functional language, loosely based on the λ-calculus. It is shown how through syntactic sugar, this language can be extended to a full language similar to Haskell [56], and providing the traditional properties of functional languages such as equational definition of functions, pattern matching, non-strict semantics, and functions as first-class values. The syntax of this full language is shown in figure 5.3. In examples of the use of $I\lambda c$ in defining language constructs, we shall feel free to use this much richer language, rather than the smaller core from which the language is defined.

In $I\lambda c$, I/O is performed using the predefined operations `read` and `write`: "`read` c" results in a value being read from the communication port c. The expression: "`write` c e" results in the value e being sent over the communication port c. From these basic primitives, it is possible to build random value generation, from which non-determinism is introduced into $I\lambda c$.


```
script ::= { eqn } expr

eqn    ::= fnName { pattern } = expr .

pattern ::= _
          | literal
          | variable
          | [ [pattern { , pattern }] ]
          | cSym { pattern }
          | ( pattern { , pattern } )

variable ::= lowerId
fnName   ::= lowerId
cSym     ::= upperId

literal ::= stringLit | intLit
          | True | False

binaryOp ::= and | or | = | -= | >= | <=
           | < | > | + | - | ++ | * | /
           | div | mod | :

unaryOp ::= hd | tl | # | not | -

ioExpn  ::= read expn
          | write expn expn
          | bind pattern = expn
                 { , pattern = expn }
            in
                 expn
            end bind
```

```
expn ::= literal
       | fnName
       | variable
       | [ [expn { , expn }] ]
       | cSym { expn }
       | ( expn { , expn } )
       | if expn then
            expn
         { elsif expn then
            expn }
         else
            expn
         end if
       | let pattern = expn
            { , pattern = expn }
         in
            expn
         end let
       | case expn of
            pattern -> expn
            { | pattern -> expn }
         end case
       | fn pattern -> expn end fn
       | expn expn
       | unaryOp expn
       | expn binaryOp expn
       | ioExpn
```

Figure 5.3: Abstract context-free syntax of a functional language built from the Interaction Lambda Calculus.

Randomness

Random values are generated by reading from the communication port `"rand"`. We place no constraints on the values read from `"rand"`, so that the communication port behaves as a random value generator. The function *rand* is defined as:

```
rand = read "rand".
```

Another useful function is *flip*, which randomly returns a value of *True* or *False*:

```
flip = (rand = 0).
```

That is, if *rand* has the value "0", then *flip* has the value *True*; otherwise, *flip* has the value *False*.

Note that there is no probability assigned to the generation of any particular values with the *rand* function. Each value generated by *rand* leads to some behaviour of the program, none of which is considered by the semantics to be any more likely than another.

The introduction of random numbers allows us to define the concept of indefinite repetition. We can define a function *repeat* which returns a list of zero or more repetitions of its argument:

```
repeat e =
    if flip then
        []
    else
        e : (repeat e)
    end if.
```

Recalling that the expression e may not be referentially transparent, this function returns a list of possibly different evaluations of e. For example, the expression:

```
repeat (read "stdin")
```

would read zero or more values from standard input, and return them in a list.

Binding Values

A basic property of $I\lambda c$ is that expressions involving I/O are not necessarily referentially transparent. This can cause difficulties when we wish to evaluate an expression, and use its result in other expressions. Consider, for example, we wish to write a program that reads in a number, and writes out whether it is positive, negative or zero. Our first attempt at this program might be:


```
let x = read "stdin" in
    if x = 0 then
        put "Zero"
    elsif x < 0 then
        put "Negative"
    else
        put "Positive"
    end if
end let
```

According to the semantics of $I\lambda c$, the variable x is bound to the *expression* "`read "stdin"`", not the value obtained by performing a read. This means that each time x is used in the body of the *let* expression, a separate read operation may be performed, potentially giving each occurrence of x a separate value. Because of this, the specification above is not guaranteed to read in one value and determine its sign.

To solve this problem, we introduce the *bind* construct that allows variables to be bound to *values*, not expressions. This can be seen as a form of strict evaluation with respect to I/O activity. Using the new notation, we can write:

```
bind x = read "stdin" in
    if x = 0 then
        put "Zero"
    elsif x < 0 then
        put "Negative"
    else
        put "Positive"
    end if
end bind
```

Here, the read operation is performed once, and the resulting value is bound to x. References to x within the *if* expression are then references to value that was read in, giving the expected semantics.

Binding implicitly sequences I/O. Any I/O specified in the binding is guaranteed to be performed before any I/O in the body of the bind construct. This allows us to define a general sequencing construct:

```
ioSeq [] = [].
ioSeq (e : es) = bind x = e in
                     x : (ioSeq es)
                 end bind.
```

The *ioSeq* function takes a list of expressions as argument, and guarantees that the I/O performed in each expression will take place in sequential order. This function will be used extensively in later definitions.

Properties of $I\lambda c$

The properties of $I\lambda c$ are presented in appendix A. These properties are informally summarized here.

$I\lambda c$ contains a pure functional subset, called *basic* $I\lambda c$. Basic $I\lambda c$ is obtained by removing all I/O operations from $I\lambda c$. (These I/O operations are represented in the *ioExpn* production in the grammar of figure 5.3.) Theorem A.3.2 demonstrates that basic $I\lambda c$ has the property of referential transparency, meaning that any expression is guaranteed to have a unique value, regardless of the context in which it is evaluated. Referential transparency, for example, guarantees that the expression "$e + e$" is equivalent to "$2 * e$", for any sub-expression e of basic $I\lambda c$. We consider this referential transparency property to be the defining property of *pure* functional languages.

Basic $I\lambda c$ is extended with the *read*, *write* and *bind* constructs presented above. Intuitively, it is clear that once $I\lambda c$ is extended with these I/O constructs, it is longer referentially transparent. This can be confirmed in that as simple an expression as "`read "stdin"`" can take on a different value each time it is evaluated. Theorem A.3.1 proves that $I\lambda c$ expressions are not in general referentially transparent.

Despite its lack of referential transparency, many of the syntactic properties associated with the λ-calculus continue to hold in $I\lambda c$. As demonstrated in section A.3, the β-rule is sound[1]; ie,

$$\texttt{fn}\ x\texttt{->}e_1\ \texttt{end fn}\ e_2 \quad \equiv \quad e_1[e_2/x]$$

That is, applying a function in x to an argument e_2 is equivalent to the function body e_1 where all free occurrences of x are replaced by e_2, for any $I\lambda c$ expressions e_1 and e_2, including those involving I/O.

Similarly, other expected properties hold, such as simplification of *if*, *case* and *let*. For example, for any $I\lambda c$ expressions e_1 and e_2, including those involving I/O, it holds that:

$$\texttt{if True then}\ e_1\ \texttt{else}\ e_2\ \texttt{end if} \quad \equiv \quad e_1$$

and that:

$$\texttt{case}\ e_1\ \texttt{of}\ x\texttt{->}e_2\ \texttt{end case} \quad \equiv \quad e_2[e_1/x]$$

If x does not appear free in e_1, it also holds that:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\texttt{end let} \quad \equiv \quad e_2[x/e_1]$$

There is no equivalent simplification rule, however, for the *bind* construct.

These properties turn out to be appropriate for the definition of extended functional languages. As we saw in chapter 2, extended functional languages are typically based on a pure functional language similar to basic $I\lambda c$. This pure language is then extended with non-functional facilities, such as continuations or dialogue combinators. The

[1]Note that in this definition we use $I\lambda c$'s syntax for λ-abstractions. A more traditional syntax for application would be "$(\lambda x\,.\,e_1)\ e_2$".

Classical Connectives		*Future-Temporal Connectives*		*Past-Temporal Connectives*	
$\neg a$	*not a*	$\bigcirc a$	*next a*	$\bullet a$	*last a*
$a \wedge b$	a *and b*	$\Diamond a$	*sometime a*	$\blacklozenge a$	*was sometime a*
$a \vee b$	a *or b*	$\Box a$	*always a*	$\blacksquare a$	*was always a*
$a \supset b$	a *implies b*				
$a \equiv b$	a *equivalent to b*	$a \lhd b$	*a precedes b*	$a \blacktriangleleft b$	*a was preceded by b*
		$a \trianglelefteq b$	*a prequals b*	$a \trianglelefteq b$	*a was prequaled b*
$\forall x\,.\,p(x)$	*for all* $x\,.\,p(x)$	$a \rhd b$	*a follows b*	$a \blacktriangleright b$	*a was followed by b*
$\exists x\,.\,p(x)$	*exists* $x\,.\,p(x)$	$a \trianglerighteq b$	*a foquals b*	$a \trianglerighteq b$	*a was foqualed b*
		$a \sqsubseteq b$	*a together b*	$a \blacksquare b$	*a was together b*

Figure 5.4: Connectives in interaction logic.

I/O features of $I\lambda c$ can be used to define these extended constructs, which are not guaranteed to possess the usual features of functional languages. When programming within the purely functional subset of the language, however, the programmer will be able to assume that the normal properties hold.

Perhaps surprisingly, the simple I/O features present in $I\lambda c$ are sufficient to model all of the features normally found in imperative languages and languages supporting concurrent programming. In section 5.3, it is shown how $I\lambda c$ can be used to model sequenced I/O, variables, non-determinism, parallel processes, synchronous and asynchronous communication and shared memory.

5.2.2 Interaction Logic

$I\lambda c$ programs provide no restrictions on the order in which I/O events take place. To express these constraints, a second level of description is used. Here, assertions are written in a temporal logic about the events that occur over communication ports. Temporal logic [36] is a modal extension of classical logic, where special temporal operators have been added. The logic presented here, *interaction logic*, is loosely based on Gabbay's USF logic [31].

Formulae in interaction logic have an implicit reference to the current time. For example it is possible for the formula

$$hungry(kjersti)$$

to be true in the current time (Kjersti is hungry now), to be false in 10 minutes, and true again in an hour. This differs from classical logic, where $hungry(kjersti)$ is either true or false, always. We can use the whole range of standard classical operators; for example:

$$hungry(kjersti) \supset eats(kjersti, cake)$$

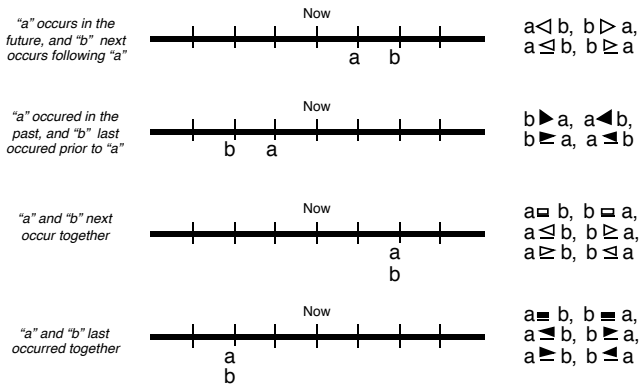


Figure 5.5: Intuitive semantics of sequencing operators – the timeline shows the order of events, and gives a set of temporal formulae that are true in the presence of these events. *Now* represents the current time.

says that whenever Kjersti is hungry, she eats some cake. This is a *constraint*: as long as in *every* time instant where $hungry(kjersti)$ holds, $eats(kjersti, cake)$ also holds, then the constraint holds over all time. Interaction logic additionally provides special temporal operators. For example,

$$\Diamond\, hungry(kjersti) \supset buy(kjersti, cake)$$

states that if sometime in the future Kjersti is going to be hungry, then Kjersti buys some cake now. (The "$\Diamond$" operator is read as *sometime*.) Temporal events can also be sequenced, as in:

$$hungry(kjersti) \supset (eats(kjersti, cake) \lhd \neg hungry(kjersti))$$

which states that if Kjersti is hungry, then she must eat some cake before she stops being hungry. (The "$\lhd$" operator is read as *precedes*.)

Interaction logic itself provides all of the facilities of classical first-order predicate logic, extended with a set of temporal operators. Figure 5.4 lists the connectives provided in interaction logic, and gives their names. As demonstrated by theorem B.3.1 in appendix B, the traditional proof rules of classical logic all apply in interaction logic as well.

In addition to the classical connectives, a rich set of temporal operators are provided.[2] Three traditional operators are provided: $\bigcirc a$ holds if a holds in the next time instant;

[2] As shown in appendix B, the core of interaction logic contains only the connectives $\bigcirc$, $\lhd$, $\bullet$, and $\blacktriangleleft$. The remaining connectives are defined from this core as syntactic sugar.

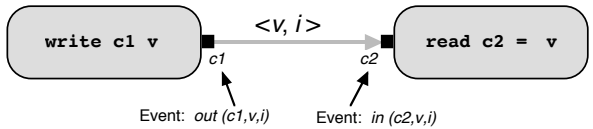


Figure 5.6: Events in TCFP – Process 1 writes value v to communication port c_1; process 2 reads value v from communication port c_2. A unique identifier i is automatically assigned to the event. At the time the value is written, $out(c_1, v, i)$ is true; at the time the value is read, $in(c_2, v, i)$ is true.

$\Diamond a$ holds if a holds sometime in the future, and $\Box a$ holds if a holds henceforth. The past-temporal reflections of these operators are provided as well – $\bullet a$ holds if a was true in the last time instant; $\blacklozenge a$ holds if a was true sometime in the past, and $\blacksquare a$ holds if a has always been true in the past.

As well as these traditional operators, interaction logic possesses operators defined specifically to accommodate reasoning about I/O sequencing. The most basic of these operators is *precedes*, defined such that $a \lhd b$ iff a occurs sometime in the future, and b does not occur until after a occurs. Note that $a \lhd b$ does not necessarily imply that b occurs at all. The past-temporal reflection of *precedes* is *was preceded by*, defined such that $a \blacktriangleleft b$ iff a occurred sometime in the past, and any occurrences of b occurred prior to the last occurrence of a.

Additional operators determine whether a and b next occur together ($a \boxminus b$), or last occurred together ($a \eqcirc b$). The *prequals* operator determines that if $a \trianglelefteq b$, then a precedes b, or occurs at the same time. $a \rhd b$ (*follows*) is the same as $b \lhd a$. Figure 5.5 shows pictorially the meanings and relationships between these operators.

These sequencing operators obey intuitive laws that makes them easy to reason with. These laws include, for example, transitivity:

$$(a \lhd b) \wedge (b \lhd c) \supset (a \lhd c)$$

distribution of negation:

$$\neg(a \lhd b) \equiv a \trianglerighteq b$$

and relation of precedes/together:

$$(a \lhd b) \vee (a \boxminus b) \equiv (a \trianglelefteq b)$$

In fact, as shown in appendix B, $\trianglelefteq$ satisfies all the axioms of a total order, and $\boxminus$ satisfies all the axioms of an equivalence relation.

5.2.3 Combining Interaction Logic and $I\lambda c$

The fundamental occurrences over which constraints in interaction logic are written are I/O events. If a value is written to a communication port at some time, then

the predefined predicate *out* will hold over that communication port and value at that time. Similarly, the predefined predicate *in* holds whenever an input event occurs over a particular communication port. I/O events are automatically assigned unique identifiers so that they can be distinguished.

Figure 5.6 shows an example of the events that are generated by reading and writing a value. If process 1 writes the value v, using the $I\lambda c$ expression "`write c1 v`", then an event occurs that at some time t_1, the value v with unique identifier i is sent out over communication port c_1. In the world of interaction logic, this means that at time t_1, the predicate $out(c_1, v, i)$ holds. Similarly, if the same value is read by process 2 at time t_2 using the $I\lambda c$ expression "`read c2`", then the predicate $in(c_2, v, i)$ holds. This notion of events provides the primary interface between $I\lambda c$ and interaction logic.

The second way in which $I\lambda c$ and interaction logic are combined is that *terms* in interaction logic are expressions in the $I\lambda c$ language. This forms a convenient way of sharing definitions between the two formalisms. For example, if we define function *hd* in $I\lambda c$ to take the first element of a list:

```
hd (x:xs) = x.
```

then we can axiomatize the function in interaction logic as follows:

$$\forall x, xs \,.\, ((hd\,(x\ :\ xs)) = x).$$

That is, for any pair of values x and xs from the $I\lambda c$ value domain, the head of x cons'd with xs is x. Here, "=" is the predefined equality predicate in interaction logic. Because $I\lambda c$ expressions are also the terms of interaction logic, it is possible to reason about $I\lambda c$ programs within the logic, thus unifying the two formalisms.

Appendix C describes the technical details of combining $I\lambda c$ and interaction logic.

Interaction logic is one-sorted, meaning all values come from the the same value domain (the value domain shared with $I\lambda c$). It can simplify some specifications, however, to use a special sort notation within interaction logic. These sorts are defined as syntactic sugar for interaction logic formulae. For example, if we wish to write that all birds fly, we write:

$$\forall x : bird \,.\, flies(x).$$

as a syntactic short-form for:

$$\forall x \,.\, (bird(x) \supset flies(x)).$$

We also use syntactic short forms to help in the definition of predicates. For example, if we wish to state that all birds are *aquatic* if they have webbed feet, we write:

$$aquatic(x : bird) \quad \equiv \quad webbedFeet(x).$$

as a syntactic short-form for:

$$\forall x : bird \,.\, (webbedFeet(x) \supset aquatic(x)).$$

Any unary predicate can be used as a sort predicate in this way. Particularly useful sort predicates are *commPort*, which holds over all values that are used to identify communication ports; *uid*, which holds over all values used as unique identifiers, and *value*, which holds over all values.

When clear from context, we allow ourselves to use this sort notation to introduce shortened versions of predicates. For example, consider we wish to specify that at any given time, a communication port can be used for either input or output, but not both. This constraint can be written as:

$$\neg\exists c : commPort \,.\, (in(c) \wedge out(c)).$$

as an abbreviation of:

$$\neg\exists c : commPort \,. \\ ((\exists v_1 : value, i_1 : uid \,.\, in(c, v_1, i_1)) \wedge (\exists v_2 : value, i_2 : uid \,.\, out(c, v_2, i_2))).$$

This section has informally introduced the syntax and semantics of TCFP. The following sections now motivate how TCFP can be used to describe interesting properties of languages. The definitions introduced in the next section will be used in defining the semantics of Clock in chapter 6.

5.3 Modelling Concurrency and Interaction in TCFP

The following sections show how TCFP can be used to model traditional forms of concurrency and interaction. The approach in this presentation is hierarchical – it is shown how more complex forms of interaction can be built from simpler forms using an inheritance-based definition style. This way, the properties and theorems pertaining to simpler forms of interaction can be reused in the more complex forms.

We first show how simple, asynchronous communication can be modeled, allowing us to model low-level network-style communication. We then add a *fifo* property to asynchronous messages, giving traditional asynchronous message passing. We then add synchronization, leading to CSP-style communication primitives. We show how variables can be implemented, leading to monitor-style shared memory. We further show how TCFP can model oracles, such as random number generators, and generators for unique identifiers.

For each style of communication, we introduce a possible language construct which is added to our functional language, and show how the construct can be modelled in TCFP.

5.3.1 Input and Output

This section introduces the basic facilities of input and output, based on the *communication port* abstraction. Figure 5.8 summarizes the new predicates and functions that

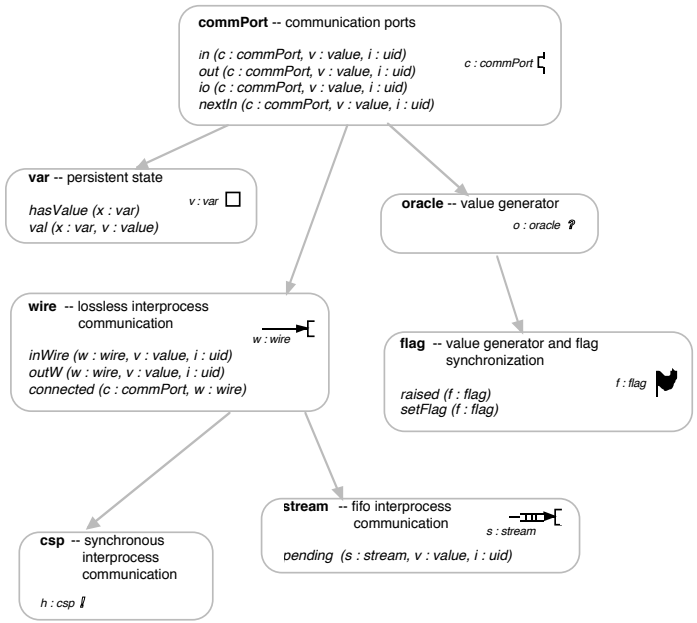


Figure 5.7: Abstractions for communications, synchronization and persistent state can be defined in TCFP. Our definition is based on inheritance, where higher-level abstractions inherit the definitions and properties of more primitive abstractions. This figure shows the abstractions defined in section 5.3; the abstractions are defined as predicates (e.g., the *stream* predicate represents fifo, lossless communication streams); predicates define operations on the abstractions (e.g., *in* and *out* hold when I/O operations take place.)

***commPort* -- Communication Ports**

New Predicates:

in (c : commPort, v : value, i : uid)	Value *v* read from commPort *c*, with uid *i*
out (c : commPort, v : value, i : uid)	Value *v* written to commPort *c*, with uid *i*
io (c : commPort, v : value, i : uid)	Value *v* read or written from/to commPort *c*, with uid *i*
in Occurs (c : commPort, v : value, i : uid)	Input event occurs either now, in the past, or in the future
outOccurs (c : commPort, v : value, i : uid)	Output event occurs either now, in the past, or in the future
ioOccurs (c : commPort, v : value, i : uid)	I/O event occurs either now, in the past, or in the future
nextIn (c : commPort, v : value, i : uid)	Next input over *c* will be value v, with uid *i*
lastIn (c : commPort, v : value, i : uid)	Last input over *c* was value v, with uid *i*
nextOut (c : commPort, v : value, i : uid)	Next output over *c* will be value v, with uid *i*
lastOut (c : commPort, v : value, i : uid)	Last output over *c* was value v, with uid *i*

New Functions:

`read c`	Read a value from communication port *c*
`write c v`	Write value *v* to communication port *c*
`rand`	Return a random value
`flip`	Return `True` of `False` randomly
`repeat e`	Return a random number of repetitions of argument *e*

Figure 5.8: Summary of the *commPort* abstraction introduced in section 5.3.1

are defined in this section.

As was shown in figure 5.6, the basis of interaction in TCFP is the *event*. Events correspond to either input or output over some communication port. Each event is automatically tagged with a unique identifier distinguishing it from all other events. In $I\lambda c$, events are generated using the predefined operations `read` and `write`. The expression "`read` c" results in a value being read from the communication port c, while the expression "`write` c e" results in the value e being sent over the communication port c. In both cases, the message generated by the `read` or `write` operation is automatically given a unique tag to distinguish it from all other messages.

In interaction logic, it is possible to reason about events using the predefined predicates *in* and *out*. The predicate $in(c, v, i)$ holds at a given time if at that time, the value v with unique identifier i is read from communication port c. The predicate $out(c, v, i)$ holds if the value v with unique identifier i is written to the communication port c. We also define the *io* predicate, which holds when either an input or an output event take place over a communication port:

$$io(c : commPort, v : value, i : uid) \equiv in(c, v, i) \vee out(c, v, i). \tag{5.1}$$

Communication Ports

All I/O events take place over some communication port. Communication ports satisfy a series of properties – for example, only one I/O event can occur over a given port at any given time, and all output events are tagged with an identifier that is guaranteed to be unique. We define these properties using constraints in interaction logic. These constraints define properties of a predicate *commPort*, which is used to identify which values are communication port names.

We define two constraints that specify that only one I/O event can take place over a given port at any given time. This is accomplished by saying that it is impossible for both an input and output event to take place over a communication port at a given time, and that whenever two I/O events take place at the same time over the same communication port, they must in fact be the same event:

$$\forall c : commPort \,.\, \neg(in(c) \wedge out(c)). \tag{5.2}$$

$$\forall c : commPort, v, v' : value, i, i' : uid \,. \tag{5.3}$$
$$io(c, v, i) \wedge io(c, v', i') \supset v = v' \wedge i = i'.$$

A series of constraints is then required to guarantee the uniqueness of tags. Informally, the rules about tags are as follows: when a value is output over a port, it is assigned a unique tag. No other output event can occur using the same tag. The tag is associated with the value written. Because of this, input events may use the same tag, but must correspond to a read of the same value written with the tag:

$$\forall c, c' : commPort, i : uid \,.\, out(c, i) \wedge out(c', i) \supset c = c'. \tag{5.4}$$

$$\forall v, v' : value, i : uid \,.\, \blacklozenge\, out(v, i) \wedge in(v', i) \supset v = v'. \tag{5.5}$$

$$\forall i : uid \,.\, out(i) \supset \neg \blacklozenge\, out(i) \wedge \neg \Diamond\, out(i). \tag{5.6}$$

This description of communication ports and their properties forms the basis of the I/O and communication mechanisms we use in TCFP. The more complex forms of interactions we define will be constructed using communication ports, and hence will inherit all of their properties and axioms.

Useful I/O Predicates

Building from the basic *in* and *out* predicates, we can define predicates over I/O events that will prove useful in later sections. First, we define predicates to specify that an I/O operation occurs at some time during the execution of the program, either in the past, the present or the future:

$$inOccurs(c : commPort, v : value, i : uid) \tag{5.7}$$
$$\equiv \blacklozenge\, in(c, v, i) \vee in(c, v, i) \vee \Diamond\, in(c, v, i).$$

$$outOccurs(c : commPort, v : value, i : uid) \tag{5.8}$$
$$\equiv \blacklozenge\, out(c, v, i) \vee out(c, v, i) \vee \Diamond\, out(c, v, i).$$

$$ioOccurs(c : commPort, v : value, i : uid) \tag{5.9}$$
$$\equiv inOccurs(c, v, i) \vee outOccurs(c, v, i).$$

The *next input* event to occur over a communication port is characterized by the fact that no input occurs on that communication port between now and the time of the input:

$$nextIn(c : commPort, v : value, i : uid) \tag{5.10}$$
$$\equiv \forall v' : value, i' : uid \,.\, (v' \neq v \vee i' \neq i) \supset (in(c, v, i) \lhd in(c, v', i')).$$

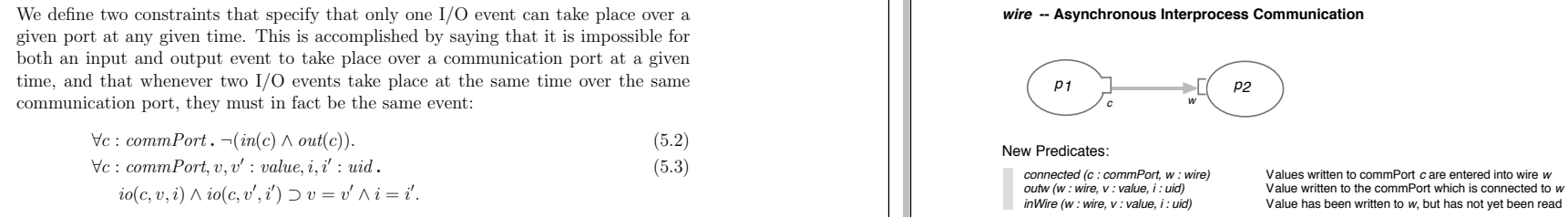


Figure 5.9: Summary of the *wire* abstraction introduced in section 5.3.2

The *last input* is defined as the last input to occur in the past:

$$lastIn(c : commPort, v : value, i : uid) \tag{5.11}$$
$$\equiv \forall v' : value, i' : uid \,.\, (v' \neq v \vee i' \neq i) \supset (in(c, v', i') \blacktriangleright in(c, v, i)).$$

The predicates *nextOut* and *lastOut* are defined similarly:

$$nextOut(c : commPort, v : value, i : uid) \tag{5.12}$$
$$\equiv \forall v' : value, i' : uid \,.\, (v' \neq v \vee i' \neq i) \supset (out(c, v, i) \lhd out(c, v', i')).$$

$$lastOut(c : commPort, v : value, i : uid) \tag{5.13}$$
$$\equiv \forall v' : value, i' : uid \,.\, (v' \neq v \vee i' \neq i) \supset (out(c, v', i') \blacktriangleright out(c, v, i)).$$

5.3.2 Processes and Asynchronous Communication

In $I\lambda c$, any expression can be a process. Since $I\lambda c$ has no specified control flow, any pair of expressions can be evaluated sequentially in any order, or concurrently. Constraints in the interaction logic can be used to reduce concurrency, but nothing special is required to introduce concurrency – it is simply there as part of $I\lambda c$.

Communication between processes is modelled via `read` and `write` events over communication ports. The properties of the ports are augmented with constraints defining the desired communication properties.

The first form of communication we will consider is simple asynchronous message passing. Intuitively, this is the form of communication one might find in a distributed systems environment, where when sending a message from one process to another, it is guaranteed that messages eventually arrive, but where the order of arrival is not guaranteed.

The new concept introduced to support this form of communication is called a *wire*. Intuitively, a wire is a communication port with a buffer. As messages arrive addressed to the communication port, they are stored in the buffer until they are read. A predicate

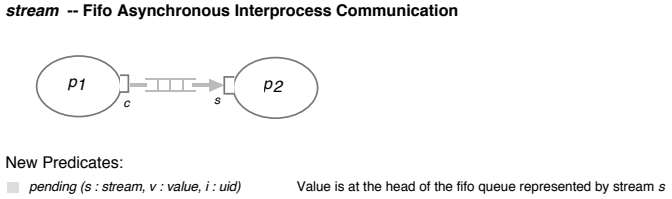


Figure 5.10: Summary of the *stream* abstraction introduced in section 5.3.3

inWire is defined that determines whether a given message has been sent but not yet received.

A second communication port can be attached to a wire, forming an asynchronous communication link. The *connected* predicate indicates whether a communication port is currently connected to a wire. A predicate *outw* is defined that specifies whether output has been performed over the communication port currently connected to the wire.

$$outw(w : wire, v : value, i : uid) \tag{5.14}$$
$$\equiv \exists c : commPort \,.\, (connected(c, w) \wedge out(c, v, i)).$$
$$\forall c, c' : commPort, w : wire \,. \tag{5.15}$$
$$connected(c, w) \wedge connected(c', w) \supset c = c'.$$
$$inWire(w : wire, v : value, i : uid) \tag{5.16}$$
$$\equiv \blacklozenge\, outw(w, v, i) \wedge \neg in(w, v, g).$$
$$\forall w : wire, v : value, i : uid \,. \tag{5.17}$$
$$in(w, v, g) \supset \bullet\, inWire(w, v, g).$$

From these definitions, two theorems can be proven. The first states that only one message can be read at a time. That is, if two messages are "in" the wire, waiting to be read, it is not possible for them to be read at the same time:

$$\forall w : wire, i, i' : uid \,.\, outw(w, i) \wedge outw(w, i') \supset i' = i. \tag{5.18}$$

The second states that a message can be read exactly once. That is, if a message is read at the current time, it can be never be read again:

$$\forall w : wire, v : value, i : uid \,.\, in(w, v, i) \supset \Box \neg inWire(w, v, i). \tag{5.19}$$

5.3.3 Stream Communication

The wire-based communication introduced in the last section had one major shortcoming – messages sent from one process to another were not guaranteed to be read in the same order that they were sent. While this form of communication adequately models low-level network communication (hence the term *wire*), we will wish to add further constraints to model higher-level communication primitives.

The *stream* abstraction augments the properties of wires to include a *fifo* property, where messages arrive in the same order they were sent. Streams introduce a new predicate, *pending*, which indicates whether a message is the next message to be read from a stream buffer. First, streams inherit all the properties of wires:

$$\forall s : stream \,.\, wire(s). \tag{5.20}$$

The *pending* predicate is defined to be true of the oldest message in the stream buffer. That is, a pending message must be in the buffer (*inWire*), and and other messages in the buffer must have arrived later than the pending message:

$$pending(s : stream, v : value, i : uid) \tag{5.21}$$
$$\equiv\ inWire(s, v, i)$$
$$\wedge \forall i' : uid \,.\, ((inWire(s, i') \wedge i \neq i') \supset (outw(s, i) \blacktriangleright outw(s, i'))).$$

Then the order of inputs from a stream is constrained to match the order of the outputs to the stream:

$$\forall s : stream, v : value, i : uid \,.\, (in(s, v, i) \supset \bullet\, pending(s, v, i)). \tag{5.22}$$

To ensure us that this definition of streams indeed matches our intuition of fifo buffers, the following theorems about streams can be proven. First, only one message can be pending on a stream at any time – that is, there can be only one oldest element:

$$\forall s : stream, v, v' : value, i, i' : uid \,. \tag{5.23}$$
$$(pending(s, v, i) \wedge pending(s, v', i') \supset v = v' \wedge i = i').$$

Secondly, removing a message from a stream means that it is no longer pending, and will never be pending again:

$$\forall s : stream, v : value, i : uid \,.\, (in(s, v, i) \supset \Box \neg pending(s, v, i)). \tag{5.24}$$

Stream-based communication is appropriate for modelling asynchronous message passing systems, such as Unix's stream sockets, or the X Client-Server communication protocol.

5.3.4 Synchronous Communication

Another popular form of process communication is *synchronous* communication. This form is embodied in theoretical calculi, such as Hoare's CSP, and in practical languages such as Ada and Occam. The key property of synchronous communication is that the send and receive operations are *synchronized* – that is, if one process sends a message to another one, the sending process must wait until the receiving process has taken the message.

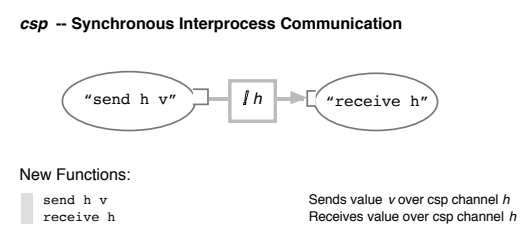


Figure 5.11: Summary of the *csp* abstraction introduced in section 5.3.4

Synchronous communication in fact combines two operations into one construct – communication and synchronization. Synchronization is particularly useful in the presence of shared resources, as it allows one process to force another one to wait until it is finished with a shared resource.

In TCFP, we model synchronous communication with the *csp* abstraction. This abstraction builds on the properties of wires, but forces a sending process to wait until the receiving process has read the message that was sent. In traditional systems, a "waiting" process performs no computation whatsoever. In TCFP, however, a waiting process is restricted only to performing no I/O. Since I/O is the only form of side-effect available in TCFP, a process that performs no I/O is indistinguishable from one that performs no computation at all.

We define that the *csp* predicate inherits all the properties of wires:

$$\forall h : csp \,.\, wire(h). \tag{5.25}$$

We then define the property that I/O over a csp channel must be synchronous. That is, a message must be received at the same time it is sent; or, a process must wait until its receiver is ready to receive before it is permitted to send a message:

$$\forall h : csp, v : value, i : uid \,.\, (outw(h, v, i) \equiv in(h, v, i)). \tag{5.26}$$

5.3.5 Variables and Shared Memory

The TCFP nomenclature of communication ports, reads and writes tends to imply that all communication must be of a message-passing style. In fact, TCFP can used to model other forms of communication, such as variables, random number generators, and oracles. Here we define how variables can be modeled.

Variables provide the facility to store a value and later retrieve it. Different processes may set and access the same variable, thereby providing shared memory between processes.

Variables are built from communication ports:

$$\forall x : var \,.\, commPort(x). \tag{5.27}$$

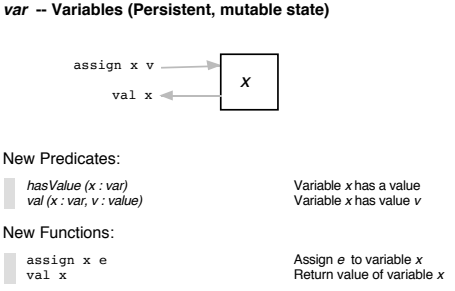


Figure 5.12: Summary of the *var* abstraction introduced in section 5.3.5

Variables may be set by writing to their communication port. Reading from a variable's communication port returns the last value set to the variable. A variable's value can be read multiple times after it has been set, corresponding to each access of the variable. This multiple-read behaviour distinguishes variables from the message passing we have seen earlier, where each message could be read only once.

We first define that a variable has a value iff a value has been set to the variable:

$$hasValue(x : var) \equiv \blacklozenge\, out(x). \tag{5.28}$$

We then define that the value of a variable to be the last value written to the variable. That is, no other value was written to the variable since the value was written:

$$val(x : var, v : value) \;\equiv\; \forall v' : value \,.\, (out(x, v') \blacktriangleright out(x, v)). \tag{5.29}$$

We then define that any value read from a variable must be the variable's value:

$$\forall x : var, v : value \,.\, (in(x, v) \supset val(x, v)). \tag{5.30}$$

From these definitions, we can prove as a theorem that if a variable has no value, it is not possible to input from it:

$$\forall x : var \,.\, \neg hasValue(x) \supset \neg\exists v : value \,.\, val(x, v). \tag{5.31}$$

It is then easy to introduce variables into a functional language. Assigning to a variable and referring to a variable's value can be accomplished through the functions *assign* and *val*, which can be defined as follows:

```
assign x e = write x e.
val x = read x.
```

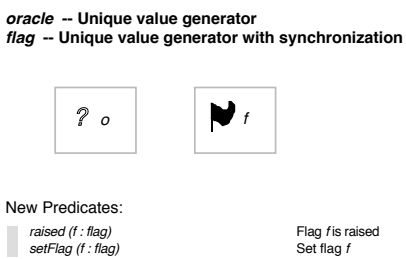


Figure 5.13: Summary of the *oracle* and *flag* abstractions introduced in section 5.3.6

5.3.6 Oracles and Flags

Sometimes within a functional program, it is necessary to obtain a value that is guaranteed to be unique. The *oracle* abstraction is provided for this purpose. Whenever a read is performed from an oracle, it is guaranteed that the value read has not been read from any other oracle. Example applications (as will be seen in chapter 6) include the assignment of a unique group identifier to a set of I/O events.

Oracles are built from communication ports:

$$\forall o : \mathit{oracle} \,.\, \mathit{commPort}(o). \tag{5.32}$$

Values read from oracles are unique:

$$\begin{aligned} &\forall o, o' : \mathit{oracle}, v : \mathit{value} \,. \\ &\quad (\mathit{in}(o, v) \supset (\neg \blacklozenge \mathit{in}(o', v) \wedge \neg \Diamond \mathit{in}(o', v) \wedge (\mathit{in}(o', v) \supset o = o'))). \end{aligned} \tag{5.33}$$

Flags are a kind of oracle supporting a blocking semantics. Similarly to oracles, reading from a flag delivers a unique value. However, it is only possible to read from a flag when it has been explicitly set. As will be seen in chapter 6, flags can be used to trigger actions such as the computation of a Clock view function.

Flags are based on oracles:

$$\forall f : \mathit{flag} \,.\, \mathit{oracle}(f). \tag{5.34}$$

A flag is *raised* if it has been set, and not yet read:

$$\mathit{raised}(f : \mathit{flag}) \quad \equiv \quad \exists v : \mathit{value} \,.\, (\mathit{setFlag}(f, v) \blacktriangleleft \mathit{in}(f)). \tag{5.35}$$

It is only possible to read from a flag if it has been raised:

$$\forall f : \mathit{flag} \,.\, (\mathit{in}(f) \supset \mathit{raised}(f)). \tag{5.36}$$

5.4 Conclusion

This section has introduced the TCFP framework. TCFP is used to specify the semantics of extended functional languages, and to provide a framework for reasoning about such extended languages.

TCFP is novel in that it combines an extended λ-calculus with a temporal logic. As was seen in the last section, the $I\lambda c$ calculus allows the introduction of I/O, processes, process communication, non-determinism and variables into functional languages, while still preserving many of the traditional properties of λ-calculus, such as the soundness of β-reduction. The interaction logic is used to restrict the possible behaviours of $I\lambda c$ programs. In the preceding sections, we have seen how interaction logic is used to introduce synchronization, process connection and sequentialization, and the mutability properties of variables. The examples have shown that TCFP's generate-and-restrict approach to defining semantics is general enough to specify all common forms of interaction and synchronization, while allowing relatively simple specifications.

Chapter 6 uses TCFP to define the semantics of Clock. Chapter 7 uses TCFP to investigate Clock's properties, and in particular, to prove the declarative properties of Clock discussed in chapter 4.

Chapter 6

Semantics of the Clock Language

We now take advantage of the TCFP framework developed in chapter 5 in order to formally define the semantics of the Clock language. These semantics fully capture the meaning of Clock programs, specifying precisely what behaviour is expected from the execution of a program.

TCFP is an appropriate formalism for expressing Clock's semantics, since the language contains non-deterministic constructs that are difficult to represent using traditional methods. For example, the language guarantees that whenever a component's view becomes out of date, the display must be updated. The language leaves it open, however, as to what order display updates may take place; similarly, an implementation is permitted to update the display more often than is strictly necessary. User inputs can be processed purely sequentially, but the language also permits concurrent implementations. The TCFP definition of Clock specifies only what must happen, and sometimes in what order. If the order is unimportant, the implementor is left free to choose. The language semantics generally follow an approach of specifying that anything can happen, as long as certain restrictions are observed. This approach maps in a straight-forward manner to TCFP's generate and restrict paradigm.

The semantic description is structured in layers. The lowest layer describes the components themselves, while the highest layer describes concurrency issues:

Layer I: Component Definition: describes the basic properties of components themselves, including what I/O ports they possess.

Layer II: Connectivity: specifies how components are connected together to form an architecture tree.

Layer III: Threads: describes how sets of I/O events are grouped together.

Layer IV: Routing: describes how updates, requests and subviews are routed.

Layer V: Triggering: specifies when invariants and views must be updated.

Layer VI: Sequencing: specifies constraints on the order in which actions may be performed.

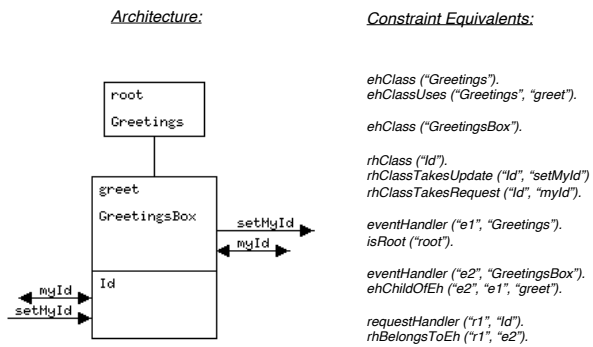


Figure 6.1: We assume that predefined predicates are available to specify the basic composition of the program's architecture. Here, it is shown how predefined predicates define a simple architecture from chapter 3. The semantics of the Clock language are built from these predefined predicates.

6.1 Layer I: Component Definition

The first step in defining the TCFP semantics of Clock is to define the basic components of Clock architectures. As we saw in chapters 3 and 4, Clock architectures are *drawn* by direct manipulation in the ClockWorks programming environment [67]. In layer I of the Clock semantics, we first show how architectures can be represented by constraints in interaction logic, and then define the TCFP semantics of event handlers and request handlers.

6.1.1 Architectures as Constraints

As it is inconvenient to directly define the semantics of an architecture that has been graphically designed, we assume that predefined predicates exist to express the structure of architecture diagrams. Figure 6.1 shows how an architecture defined in chapter 3 is mapped into architecture constraints based on these predefined predicates. Since the translation from architecture diagrams to constraints is straightforward, we will simply assume that for any given architecture, the constraints are predefined.[1]

[1]It is interesting to note that these constraints are not only of theoretical interest. ClockWorks uses the constraint format to save architectures in a file. The Clock compiler uses this constraints file to implement the architecture.

Predicates Representing Classes

The *ClockWorks* environment is based on a *prototype-instance* model of classes [72], where any component instance can also be used as a class. In the Clock semantics, we make the distinction between class and instance more clear – the name given to each component in the architecture diagram is assumed to be the component's class. A unique name must then be assigned to each instance of the class. For example, in figure 6.1, *Greetings*, *GreetingsBox* and *Id* are all class names. It is not necessary to show the name of the *instance* of these components, since the architecture diagram itself already distinguishes the components.

The second name shown in the architecture diagram is a *subview* name, which an event handler's parent uses to refer to the event handler's view. The subview names are part of the class definition of the parent. For example, in figure 6.1, the *Greetings* event handler refers to the view of the *GreetingsBox* by the subview name *greet*.

We assume that the predicate *ehClass* is defined so that $ehClass(ec)$ holds iff ec is an event handler class; similarly, $rhClass(rc)$ holds iff rc is an request handler class. The predicate $request(rq)$ holds iff rq is a request, and $update(u)$ holds iff u is an update.

The interfaces of component classes are defined through the predicates:

$$ehClassTakesUpdate(ec : ehClass, u : update)$$
$$rhClassTakesUpdate(rc : rhClass, u : update)$$
$$rhClassTakesRequest(rc : rhClass, rq : request)$$

where $ehClassTakesUpdate\ (ec, u)$ holds iff ec handles the input u, $rhClassTakesUpdate\ (ec, u)$ holds iff rc handles the update u, and $rhClassTakesRequest\ (rc, rq)$ holds iff rc handles the request rq. As can be seen in figure 6.1, the definition of these predicates comes straight from the architecture diagram drawn by the user.

The final part of the interface of event handler components is what subviews the component may use. A predicate is assumed to be defined from the subview definition of an event handler class, so that if event handler class ec uses a subview sv, then it holds that $ehClassUses(ec, sv)$. From this, we define a predicate *svid* identifying all subview names:

$$svid(sv) \quad \equiv \quad \exists ec : ehClass \,.\, ehClassUses(ec, sv). \tag{6.1}$$

Predicates Representing Components

Components are instantiated from classes. We assume that predefined predicates exist to specify what components exist within the architecture. In particular, we assume the predicate $eventHandler(e, ec)$ holds iff the event handler e has been declared to be of class ec, and that $requestHandler(r, rc)$ holds iff the request handler r has been declared to be of class rc.

We can then define some useful predicates describing components. The *eh* and *rh* predicates hold over all event and request handler instances respectively; these predicates are particularly useful as sort predicates. The *component* predicate identifies all

components, i.e., all event and request handlers:

$$eh(e) \equiv \exists ec : ehClass\,.\,eventHandler(e, ec). \quad (6.2)$$
$$rh(r) \equiv \exists rc : rhClass\,.\,requestHandler(r, rc). \quad (6.3)$$
$$component(c) \equiv eh(c) \vee rh(c). \quad (6.4)$$

The request/update interface of a component is the same as the interface of the component's class. We define predicates to describe components' interfaces based on the equivalent predicates over classes:

$$ehTakesUpdate(e : eh, u : update) \quad (6.5)$$
$$\equiv \exists ec : ehClass\,.\,eventHandler(e, ec) \wedge ehClassTakesUpdate(ec, u).$$
$$ehTakesSubview(e : eh, sv : svid) \quad (6.6)$$
$$\equiv \exists ec : ehClass.eventHandler(e, ec) \wedge ehClassUses(ec, sv).$$
$$rhTakesUpdate(r : rh, u : update) \quad (6.7)$$
$$\equiv \exists rc : rhClass.requestHandler(r, rc) \wedge rhClassTakesUpdate(rc, u).$$
$$rhTakesRequest(r : rh, rq : request) \quad (6.8)$$
$$\equiv \exists rc : rhClass.requestHandler(r, rc) \wedge rhClassTakesRequest(rc, rq).$$

Predicates to Express Architecture Structure

The structure of an architecture is defined using the predefined predicates *rhBelongsToEh*, *ehChildOfEh*, and *root*. As seen in figure 6.1, the definition of these components comes straight from the architecture diagram.

The predicate *rhBelongsToEh* specifies where request handlers are placed in the architecture. For example, in figure 6.1, *rhBelongsToEh*(*"r1", "e2"*). If a component e_c is a child of component e_p, and if e_p refers to this component's subview as sv, then $ehChildOfEh(e_c, e_p, sv)$ holds. In figure 6.1, *ehChildOfEh*(*"e2", "e1", "greet"*). Finally, if an event handler e is the root of the the tree, then $root(e)$ holds. In figure 6.1, $root(e1)$ holds, identifying the *Greetings* component as the root of the tree.

6.1.2 Semantics of Event Handlers

Following the *generate and restrict* approach of TCFP, components are specified in two parts – an $I\lambda c$ function generates a set of possible behaviours for the component, and constraints in interaction logic restrict the behaviours to the correct ones.

Event handlers are built from the TCFP abstractions developed in chapter 5. As shown in figure 6.2, event handlers are built from communication ports, streams, wires, variables, flags and oracles. The exact function of all of these parts will be made clear in the following sections. For now, we provide an overview of the definition.

For simplicity's sake, we assume the existence of functions giving unique names to each of the ports used in an event handler. For example, "*updtIn e*" is the input-port for updates sent to event handler e.

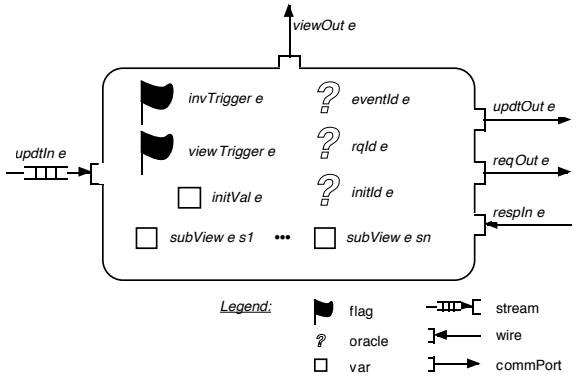


Figure 6.2: The TCFP definition of an event handler component e. The event handler takes inputs on an *updtIn* port. Updates are sent on an *updtOut* port; requests are sent on a *reqOut* port, and responses are returned on a *respIn* port. The final view is delivered on the *viewOut* port. The *invTrigger* and *viewTrigger* are flags used to determine when the invariant and view functions must be updated. A series of subview variables is used to represent the values of any subviews the event handler may have.

The basic function of an event handler is to provide a view and to respond to input. Whenever the component has a new view to report, it is written to the port "*viewOute*". Whenever a new input arrives at the component, it is queued in the stream "*updtIn e*" for fifo processing.

Event handler functions may require that updates and requests be sent to components higher up the tree. These updates and requests are sent out over the communication ports "*updtOute*" and "*reqOute*" respectively. Responses returned from requests arrive in the wire "*respIn e*". Note that the use of a *wire* for responses implies that responses are not necessarily received in the order in which they were sent.

An event handler's *view* function may require the views of children of the component; these childrens' views are referred to as *subviews*. The subview of each child s is stored in a variable "*subView e s*".

The initializing value of the component is stored in a variable "*initVal e*", for use in the components *initially* function.

As is explained in section 6.3, sets of I/O events are logically grouped into *threads* according to their purpose. For example, the I/O used to process a single request/response pair is grouped into a request thread, so that the response can be correctly matched to the request. Similarly, all I/O used to compute a view function is grouped into a *view* thread. Identifiers for these threads are obtained from a set of oracles and flags (see sections 5.3.6 and 5.3.6.) For example, the oracle "*rqId e*" assigns identifiers to request threads; the oracle "*eventId e*" assigns identifiers to input threads.

Flags are used to assign identifiers to invariant and view threads. This allows us to explicitly trigger the start of these threads when they are required – for example, triggering a view update when a component's view is out of date. This use of triggers is detailed in section 6.5.

The structure of event handlers, as shown in figure 6.2, is specified as follows:

$$
\begin{aligned}
&\forall e : eh\,\textbf{.} && (6.9)\\
&\quad stream(updtIn\,e)\\
&\wedge\quad commPort(updtOut\,e)\\
&\wedge\quad commPort(reqOut\,e) \wedge wire(respIn\,e)\\
&\wedge\quad commPort(viewOut\,e)\\
&\wedge\quad oracle(eventId\,e) \wedge oracle(rqId\,e) \wedge oracle(initId\,e)\\
&\wedge\quad flag(invTrigger\,e) \wedge flag(viewTrigger\,e)\\
&\wedge\quad var(initVal\,e)\\
&\wedge\quad \forall s\,\textbf{.}\ ehTakesSubview(e, s) \supset var(subView\,e\,s).
\end{aligned}
$$

The $I\lambda c$ code for event handlers is shown in figure 6.3. The function *mkEHC* makes an event handler class. The *mkEHC* function takes four functions as parameters, the *event*, *invariant*, *initially* and *view* functions provided by the Clock programmer. The resulting event handler class function can be applied to an event handler name to instantiate an event handler component.

The details of the $I\lambda c$ code are explained throughout the remaining sections of this chapter. The basic structure, however is as follows. The main body of the function can be stated in pseudo-code as:

```
repeat oneOf [
    initialize, processNextInput, doInvariant, updateView
]
```

That is, the component performs an indefinite series of initializations, inputs, invariants and view updates. There is nothing in the $I\lambda c$ code to specify such obvious constraints as that the initialization should be performed only once, before any other computation begins, or that the view updates must be performed whenever the view changes – such constraints are encoded in interaction logic, and will appear in later sections.

All I/O is performed over the ports that were defined in figure 6.2. For example, view updates are issued by sending them over the port "*viewOut e*"; in figure 6.3, this behaviour is encoded in the definition:

```
sendView = write (viewOut e).
```

Similarly, updates are written to the port "*updtOut e*", as captured in the definition of *sentUpdt*.

Both the *initially* and *view* functions may issue requests, and therefore take a request function as parameter. This function, *requestFn*, is defined to write a given request to


```
mkEHC eventFn invariantFn initiallyFn viewFn =
    fn e ->
        let eventThread = read (eventId e),
            invariantThread = read (invariantTrigger e),
            viewThread = read (viewTrigger e),
            initThread = read (initId e),
            requestThread = read (rqId e),

            requestFn rq =
                group requestThread
                    bind s = write (reqOut e) rq in
                        read reqIn
                    end bind,

            subviewFn svName = val (subview e svName),

            sendUpdt = write (updtOut e),
            sendView = write (viewOut e),

            handleEvent = eventFn requestFn,
            doInvariant = invariantFn requestFn,
            doInitially = initiallyFn requestFn,
            newView = viewFn requestFn subviewFn
        in
            repeat oneOf [
               let initialValue = val (initVal e) in
                   group initThread (
                       sendUpdt (initiallyFn initialValue)
                   )
               end let,

               let nextEvent = read (updtIn e) in
                   group eventThread (
                       sendUpdt (eventFn nextEvent)
                   )
               end let,

               group invariantThread (
                   sendUpdt doInvariant
               )

               group viewThread (
                   sendView newView
               )
            ]
        end let
    end fn.
```

Figure 6.3: An $I\lambda c$ function specifying the behaviour of event handlers. When combined with the temporal constraints represented in figure 6.2, this function gives the semantics of event handler components.

the *reqOut e* port, and read the response from the *reqIn e* port. Section 6.4.2 describes how it is guaranteed that the correct response is always read for a given request.

The *group* function is used to group together I/O activity into threads. For example, the execution of the *view* function is grouped into a view thread. Reading the thread id from the "*viewTrigger e*" flag is used to guarantee that views are updated at the correct time (as defined in section 6.5.)

The values of subviews are read from the appropriate subview variable. A subview function is defined to read a given subview:

```
subviewFn svName = val (subview e svName).
```

That is, the function reads the subview value from the appropriate variable. This subview function is passed as a parameter to the *view* function. Section 6.4.1 shows how the views of child components are connected to subview variables.

The *IλC* definition of figure 6.3 together with the temporal constraints shown in figure 6.2 give the basic definitions of event handler components. These definitions do not specify the complete behaviour of Clock programs – for example, it is specified that after issuing a request, the event handler reads some response, but not *what* response. Further connectivity and routing constraints are required to specify this. Similarly, the definitions imply that an event handler generates views, but not *when* – further triggering and sequencing constraints will answer these questions also.

Figure 6.2 shows that each event handler has a variable representing its view. An architecture also has a variable representing the view of the entire architecture tree. This view is represented in a special variable `"rootView"`:

$$var(\texttt{"rootView"}). \tag{6.10}$$

Section 6.4.1 shows how the value of this view is set.

6.1.3 Semantics of Request Handlers

The semantics of request handlers are structured in a similar way to those of event handlers. Figure 6.4 shows the structure of the interaction logic description of a request handler. Updates and requests arrive on the streams "*updtIn r*' and "*reqIn r*" for fifo processing. Responses are returned via the communication port "*respOut r*".

The current state of the request handle is represented in the variable "*state r*". The oracles "*rqId r*" and "*updtId r*" are used to group the I/O used to perform a particular request or update into a thread. The flag "*initTrigger e*" is used to group the I/O used to process the request handler's *initially* function, and to determine when the request handler should be initialized.

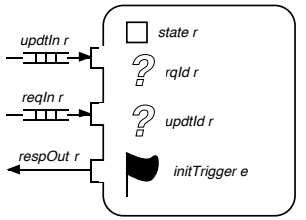


Figure 6.4: The TCFP representation of a request handler r. The state of the request handler is maintained in a *state* variable. Updates and requests are received from *updtIn* and *reqIn* ports respectively; responses are returned via a *respOut* port. (For legend see figure 6.2.)

The structure of figure 6.4 can be formalized with the following constraints:

$$\begin{aligned} &\forall r : rh\,\texttt{.} \\ &\quad stream(updtIn\,r) \\ &\wedge\ stream(reqIn\,r) \wedge commPort(respOut\,r) \\ &\wedge\ var(state\,r) \\ &\wedge\ oracle(rqId\,r) \wedge oracle(updtId\,r) \\ &\wedge\ flag(initTrigger\,r). \end{aligned} \tag{6.11}$$

The *IλC* code implementing a request handler class is found in figure 6.5. The *mkRHC* function takes three functions as parameters, the *request*, *update* and *initially* functions written by the user. The resulting function takes a request handler name as parameter, and creates a request handler instance. In pseudo-code, the basic structure of *mkRHC* is the following:

```
repeat oneOf [
    handleRequest, handleUpdate, initialize
]
```

That is, the function indefinitely handles updates, requests, and initializes itself.

The *requestFn* parameter is a function that takes the current state and a request as parameters, and returns a response. To evaluate a request, therefore, *mkRHC* reads a request (with the expression "`read (reqIn r)`"), reads the current value of the state variable ("`val (state r)`"), and writes the response to the response port (using the expression "`write (respOut r)`").

Similarly, updates are performed by applying the *updateFn* parameter to the current state and update, and assigning the new state back to the state variable (with the expression "`assign state newState`"). Initialization is performed by assigning the initial state to the state variable ("`assign state initialState`").

```
mkRHC requestFn updateFn initialState =
    fn r ->
        let requestThread = read (rqId r),
            updateThread = read (updtId r),
            currentState = val (state r),
            initiallyThread = read (initTrigger r)
        in
            repeat oneOf [
               let rq = read (reqIn e),
                   sendResponse = write (respOut r)
               in
                   group requestThread (
                       sendResponse (requestFn currentState rq)
                   )
               end let,

               let updt = read (updtIn r) in
                   group updateThread
                       let newState = updateFn currentState updt in
                           assign state newState
                       end let
               end let,

               group initiallyThread (
                   assign state initialState
               )
            ]
        end let
    end fn.
```

Figure 6.5: Code implementing a request handler class.

The oracles and flags *rqId*, *updtId*, and *initTrigger* are used to group together sets of I/O activities into threads. As will be seen in section 6.5, the *initTrigger* flag is used to make sure that the request handler is initialized before it is used.

6.2 Layer II: Connectivity

The last section specified the basic structure of Clock components, and provided predicates to state what components exist to make up an architecture. When specifying architectures, Clock programmers also specify how components are connected together – which components are children of other components, and where request handlers fit. This section defines the concept of a *well-formed* architecture, through constraints restricting how components may be connected. If these constraints are satisfied, the architecture is guaranteed to be well-formed. The constraints express restrictions that, for example, architectures must be tree structured, have finite depth and breadth, that subviews must be connected to their children, and so forth. In fact, every architecture that can be created within *ClockWorks* satisfies these well-formedness constraints. An architecture is well-formed if the following constraints hold:

An architecture has one root:

$$\exists e : eh \,.\, (root(e) \wedge \forall e' : eh \,.\, (root(e') \supset e' = e)). \tag{6.12}$$

Every event handler is either the root, or has one parent:

$$\begin{aligned} \forall e_c : eh \,.\, & (root(e_c) \wedge \neg\exists e_p : eh \,.\, ehChildOfEh(e_c, e_p) \\ & \vee \neg root(e_c) \wedge \exists e_p : eh \,.\, (ehChildOfEh(e_c, e_p) \\ & \quad \wedge \forall e'_p : eh \,.\, (ehChildOfEh(e_c, e'_p) \supset e'_p = e_p))). \end{aligned} \tag{6.13}$$

Every request handler belongs to one event handler:

$$\begin{aligned} \forall r : rh \,.\, \exists e : eh \,. \\ (rhBelongsToEh(r, e) \wedge \forall e' : eh \,.\, (rhBelongsToEh(r, e') \supset e' = e)). \end{aligned} \tag{6.14}$$

If an event handler has multiple local request handlers, their request/update interfaces must be disjoint:

$$\begin{aligned} \forall r_1, r_2 : rh, e : eh \,. \\ rhBelongsToEh(r_1, e) \wedge rhBelongsToEh(r_2, e) \\ \supset \neg\exists rq : request \,.\, (rhTakesRequest(r_1, rq) \wedge rhTakesRequest(r_2, rq)). \end{aligned} \tag{6.15}$$

For every subview of an event handler, there is one child:

$$\begin{aligned} \forall e_p : eh, sv : svid \,.\, & ehTakesSubview(e_p, sv) \\ & \supset \exists e_c : eh \,.\, (ehChildOfEh(e_c, e_p, sv) \\ & \quad \wedge \forall e'_c : eh \,.\, (ehChildOfEh(e'_c, e_p, sv) \supset e'_c = e_c)). \end{aligned} \tag{6.16}$$

It finally remains to require that all the event handlers are arranged in a single tree structure, where $root(e)$ holds of the tree root e. We first defined a predicate to span the tree:

$$inTree(e_c : eh) \quad \equiv \quad root(e_c) \vee \exists e_p : eh \,.\, (ehChildOfEh(e_c, e_p) \wedge inTree(e_p)). \tag{6.17}$$

We then insist that all event handlers be reachable by spanning the tree in this manner:

$$\forall e : eh \,.\, inTree(e). \tag{6.18}$$

Finally, we wish to constrain legal trees to being finite. That is, each event handler can have only a finite number of children, and each path through the tree has finite depth.

We assume that a predicate *ehNum* exists that assigns a natural number identifier to each event handler e. We then express the finiteness of the tree by specifying that there is an upper bound on these distinct identifiers. We first specify that each event handler has a natural number associated with it, and that these identifiers are distinct:

$$\forall e : eh \,.\, \exists n : I\!N \,.\, ehNum(e, n). \tag{6.19}$$

$$\begin{aligned} \forall e_1, e_2 : eh, n_1, n_2 : I\!N \,. \\ ehNum(e_1, n_1) \wedge ehNum(e_1, n_2) \wedge e_1 \neq e_2 \\ \supset n_1 \neq n_2. \end{aligned} \tag{6.20}$$

We then specify that there is an upper bound on the identifiers:

$$\exists b : I\!N \,.\, \forall e : eh \,.\, (ehNum(e, n) \supset n \leq b). \tag{6.21}$$

6.3 Layer III: Threads

In the Clock semantics, it is important to be able to consider related I/O activity to be grouped together. For example, a request issued by an event handler is related to the response that eventually is returned. The set of requests contributing to a single view computation are also form a related group. To allow us to express such relations, the concept of *thread* is introduced. Threads are defined to group together sets of related I/O events. In section 6.4, threads will be used to make sure that responses to requests are delivered to the right place, and in section 6.6, threads will be used to make sure that updates are processed in a correct order.

Threads are based on *oracles* and *flags* – for each thread, an oracle is used to provide a unique identifier for the thread. For example, I/O resulting from view generation is grouped into a thread using the following code (taken from figure 6.3):

```
group viewThread (sendView newView)
```

where

```
viewThread = read (viewTrigger e)
```

That is, all I/O activity resulting from the computation of the new view is grouped into the thread *viewThread*, the value of which is read from the flag *viewTrigger*. (Oracles and flags were described in section 5.3.6.)

In order to reason about threads within interaction logic, we define four predicates. First, we define that a thread is a value read from an oracle (that is, a value used to identify a thread):

$$thread(t : value) \quad \equiv \quad \exists o : oracle \,.\, inOccurs(o, t). \tag{6.22}$$

We can determine whether a particular I/O event belongs to a thread by determining whether an I/O event occurs within the group identified by the thread:

$$inThread(t : thread, i : uid) \equiv \exists c : commPort, v : value \,.\, inGroup(c, v, i, t). \tag{6.23}$$

We then define a predicate indicating whether a particular thread was issued from a given oracle:

$$threadFrom(t : thread, o : oracle) \quad \equiv \quad inOccurs(o, t). \tag{6.24}$$

This then allows us to determine whether two I/O events belong to the same thread:

$$\begin{aligned} &sameThread(o : oracle, i_1, i_2 : uid) \\ &\quad \equiv \quad \exists t : thread \,.\, (threadFrom(t, o) \wedge inThread(t, i_1) \wedge inThread(t, i_2)). \end{aligned} \tag{6.25}$$

The *sameThread* predicate allows us to determine whether two I/O events belong to the same thread, as initiated by a particular oracle. For example, we can write:

$$sameThread(viewOut\, e, i_1, i_2).$$

to determine whether two I/O events belong to the same view generation thread of some event hander e.

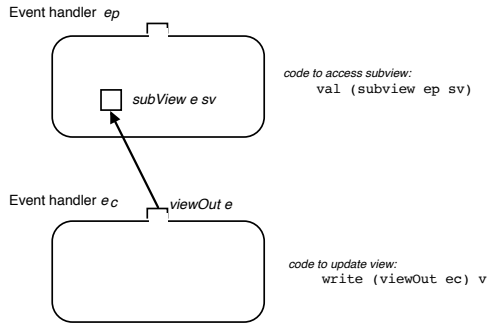


Figure 6.6: Event handler e_p can access the subview of its child e_c, connected as subview sv, by referring to the variable $subView\, e_p\, sv$. When e_c updates its view, the update is set in the variable.

6.3.1 Kinds of Threads

We define different kinds of threads for different tasks – a *view* thread handles a view update; an *input* thread handles an input, and so forth. We specify that a thread is an *input* thread if it received its thread identifier from an input oracle:

$$inputThread(t : thread) \quad \equiv \quad \exists e : eh \,.\, inOccurs(eventId\, e, t). \tag{6.26}$$

Similarly, invariant, view and initialization threads are identified by their oracles or flags:

$$invThread(t : thread) \quad \equiv \quad \exists e : eh \,.\, inOccurs(invTrigger\, e, t). \tag{6.27}$$

$$viewThread(t : thread) \quad \equiv \quad \exists e : eh \,.\, inOccurs(viewTrigger\, e, t). \tag{6.28}$$

$$initThread(t : thread) \quad \equiv \quad \exists e : eh \,.\, inOccurs(initId\, e, t). \tag{6.29}$$

6.4 Layer IV: Routing

Section 6.1 provided the basic definition of event and request handler components, while section 6.2 showed how components are joined together to form an architecture. This section defines how components communication with each other, or how information is *routed* between components.

There are two basic mechanisms which components use to communicate with each other. The first is the subview mechanism, allowing a parent component to use the views of its children when creating its own view. The second is requests and updates, allowing components to query or modify state further up the tree.

6.4.1 Subview Routing

In Clock, components may use the views of their children in making up their own views. The views of children are called *subviews*. In our semantics, subviews are represented as variables. Whenever a child component updates its view, it writes the view to a subview variable represented in its parent. Whenever the parent computes its own view, it simply refers to the current values of the subview variables. As shown in figure 6.3, if event handler e wishes to refer to its subview sv, it refers to the variable as "`val (subview e sv)`".

Figure 6.6 shows how a child component writes its subview to its parent. The *viewOut* port of the child is bound to the subview variable of the parent, so that whenever the child updates its view, the subview variable is updated:

$$\forall e_p, e_c : eh, sv : svid, v : value \,.\, (ehChildOfEh(e_c, e_p, sv) \wedge out(viewOut\, e, v) \equiv out(subView\, e\, sv, v)). \tag{6.30}$$

The view for the entire tree is represented in the variable `"rootView"`. The view of the root component is attached to this variable:

$$\forall e : eh \,.\, root(e) \wedge out(viewOut\, e, v) \equiv out(\texttt{"rootView"}) \tag{6.31}$$

6.4.2 Request/Update Routing

Event handler components can query state in request handlers appearing above them in the tree, and can issue updates which may be handled by event or request handlers appearing above them in the tree. A request or update issued by an event handler is handled by the first component appearing above the event handler that is capable of handling it. To capture this form of routing, we first define what it means for one component to be *above* another:

$$\begin{aligned} &above(c_1, c_2 : component) \\ &\quad \equiv \; ehChildOfEh(c_2, c_1) \vee rhBelongsToEh(c_2, c_1) \\ &\qquad \vee \exists e : eh \,.\, (ehChildOfEh(e, c_1) \wedge rhBelongsToEh(c_2, e)) \\ &\qquad \vee \exists c' : component \,.\, (above(c_1, c') \wedge above(c', c_2)). \end{aligned} \tag{6.32}$$

That is, as shown in figure 6.7, a parent event handler appears above its children, request handlers appear above their owning event handler, and the relation is transitively extended. We will also find it useful to express that a component c appears *between* components c_1 and c_2 in the tree:

$$between(c, c_1, c_2 : component) \;\equiv\; above(c, c_1) \wedge above(c_2, c). \tag{6.33}$$

We can then succinctly express request/update routing by stating that a request or update issued by a component e is handled by the first component c appearing above

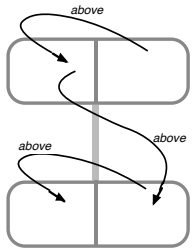


Figure 6.7: The *above* predicate captures the visibility path of updates and requests as they move up the tree.

e that is capable of handling the request/update. For requests, this is stated as:

$$\begin{aligned} &handlesRequest(r : rh, rq : request, e : eh) \\ &\quad \equiv \; rhTakesRequest(r, rq) \wedge above(r, e) \\ &\qquad \wedge \neg \exists r' : rh \,.\, (rhTakesRequest(r', rq) \wedge between(r', e, r)). \end{aligned} \tag{6.34}$$

Updates can be handled by both event and request handlers:

$$takesUpdate(c : component) \;\equiv\; ehTakesUpdate(c) \vee rhTakesUpdate(c). \tag{6.35}$$

An update is handled by the first component appearing above the issuing event handler that is capable of handing it:

$$\begin{aligned} &handlesUpdate(c : component, u : update, e : eh) \\ &\quad \equiv \; takesUpdate(c, u) \wedge above(c, e) \\ &\qquad \wedge \neg \exists c' : component \,.\, (takesUpdate(c', u) \wedge between(c', e, c)). \end{aligned} \tag{6.36}$$

Routing Requests and Updates

The predicates *handlesRequest* and *handlesUpdate* establish which components are responsible for handling the requests and updates generated by a particular event handler. Based on these predicates, we now establish constraints specifying that when a component issues a request or update, the correct component receives and handles it.

As shown in figure 6.8, this means that whenever an event handler issues an update, the update must be sent to the *updtIn* port of whatever component handles the update:

$$\begin{aligned} &\forall e : eh, u : update, c : component \,. \\ &\quad out(updtOut\, e, u) \wedge handlesUpdate(c, u, e) \\ &\qquad \supset connected(updtOut\, e, updtIn\, c). \end{aligned} \tag{6.37}$$

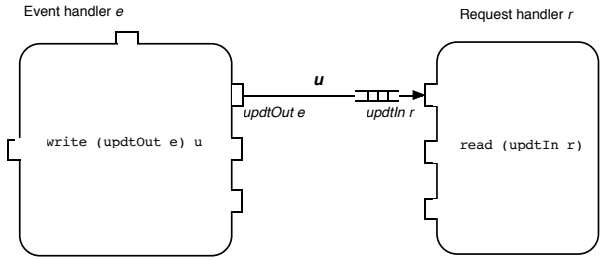


Figure 6.8: An update u issued by event handler e is written to the port $updtOut\,e$. If request handler r handles this update, then $updtOut\,e$ is connected to the stream $updtIn\,r$, and a future read from $updtIn\,r$ will obtain the update u.

Similarly, whenever an event handler e issues a request rq, the request must be sent to the *reqIn* port of whatever request handler r handles that request. This is achieved by connecting the *reqOut* port of e to the *reqIn* port of r:

$$\begin{aligned}&\forall e : eh, rq : request, r : rh\,\boldsymbol{.} \\ &\quad out(reqOut\,e, u) \wedge handlesRequest(r, u, e) \\ &\quad\quad \supset connected(reqOut\,e, reqIn\,r).\end{aligned} \tag{6.38}$$

As shown in figure 6.9, the response to a request must be written back by connecting the *respOut* port of r to the *respIn* port of e. Before specifying this condition, we first specify what it means for a response to be associated with a request. As shown in figure 6.9, when a request handler r responds to a request, it groups the reading of the request and the writing of the response into a thread. The thread identifier is obtained from the oracle $rqId\,r$. Similarly, when an event handler e issues a request, the request and response are grouped with a thread obtained from the oracle $rqId\,e$. This grouping allows us to express in interaction logic whether a response results from a given request:

$$respondsTo(i_1, i_2 : uid, c : component) \;\equiv\; sameThread(rqId\,c, i_1, i_2). \tag{6.39}$$

We can then specify that when a request handler r responds to a request issued by event handler e, then the response is sent to e:

$$\begin{aligned}&\forall r : rh, i_1, i_2 : uid, e : eh\,\boldsymbol{.} \\ &\quad \blacklozenge\, out(reqOut\,e, i_1) \wedge out(respOut\,r, i_2) \wedge respondsTo(i_2, i_1, r) \\ &\quad\quad \supset connected(respOut\,r, respIn\,e).\end{aligned} \tag{6.40}$$

Finally, we must ensure that when an event handler reads a response, that the response is matched to the correct request. This is accomplished by specifying that whenever a request handler specifies that a request/response pair are in the same thread, the

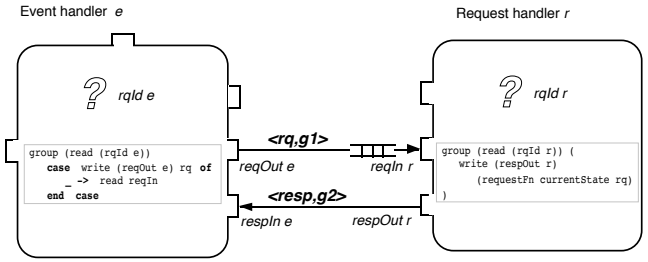


Figure 6.9: A request rq (with unique id i_1) issued by event handler e is written to the port $reqOut\,e$. If request handler r handles this update, then $reqOut\,e$ is connected to the stream $reqIn\,r$, and a future read from $reqIn\,r$ will obtain the request rq. The response $resp$ is written to $respOut\,r$, and received by e over the port $respIn\,e$. Since request and response pairs are grouped, it holds that: $sameThread(rqId\,e, i_1, i_2)$, and $sameThread(rqId\,r, i_1, i_2)$.

event handler that issued the request must also consider the response to be in the same thread.

$$\begin{aligned}&\forall r : rh, i_1, i_2 : uid, e : eh\,\boldsymbol{.} \\ &\quad outOccurs(respOut\,e, i_1) \wedge inOccurs(respIn\,e, i_2) \wedge respondsTo(i_2, i_1, r) \\ &\quad\quad \supset respondsTo(i_2, i_1, e).\end{aligned} \tag{6.41}$$

6.4.3 Routing Restrictions

An important correctness condition on Clock architectures is that any updates performed in an initialization or invariant must be *local* updates; that is, the updates must be handled by a local request handler.

We first define predicates to indicate whether a thread t makes a request or update to a request handler r:

$$\begin{aligned}&threadRequests(t : thread, r : rh) \\ &\quad\equiv\; \exists i : uid\,\boldsymbol{.}\,(in(reqIn\,r, i) \wedge inThread(i, t)).\end{aligned} \tag{6.42}$$

$$\begin{aligned}&threadUpdates(t : thread, r : rh) \\ &\quad\equiv\; \exists i : uid\,\boldsymbol{.}\,(in(updtIn\,r, i) \wedge inThread(i, t)).\end{aligned} \tag{6.43}$$

$$\begin{aligned}&threadViews(t : thread, r : rh) \\ &\quad\equiv\; \exists i : uid\,\boldsymbol{.}\,(viewVar(v) \wedge in(v, i) \wedge inThread(i, t)).\end{aligned} \tag{6.44}$$

We then specify that if a thread is an invariant or initially thread from some event handler e, then any updates must be routed to request handlers owned by e:

$$\begin{aligned}&\forall t : thread, e : eh, r : rh \,.\\ &\quad ((threadFrom(initTrigger\, e, t) \vee threadFrom(invTrigger\, e, t))\\ &\qquad \wedge threadUpdates(t, r)\\ &\qquad\quad \supset rhBelongsToEh(r, e)).\end{aligned} \tag{6.45}$$

6.5 Layer V: Triggering

The previous sections have established how architectures are constructed, and how requests and updates are routed when they are issued. It has been shown how sets of I/O are grouped into threads working towards, for example, the evaluation of a view, or the processing of an invariant. What remains to be addressed is the question of *when* views must be updated, and when invariants must be processed.

Intuitively, a view must be processed whenever it is out of date. We express this with a constraint stating that if evaluating a view would result in a new view, a view thread should be initiated. Similarly, an invariant function must be evaluated whenever the component's local state is out of date. We express this condition with a constraint stating that if evaluating an invariant were to modify the state of some request handler, then an invariant thread should be initiated. Threads to initialize components must also be initiated immediately following the creation of a component.

As was shown in figures 6.2 and 6.4, components have a set of *flags* used to trigger the evaluation view, invariant and initialization computations. The flags are a special form of oracle, providing a thread identifier used to group I/O events. Additionally, flags act as triggers for threads – only when the predicate $raised(f)$ holds is it possible to read from the flag f. Therefore, triggering a thread establishes that the thread will be executed either now or sometime in the future.

Naturally, it is not enough to state only when it is necessary to update a view or invariant, since the updates of multiple components may conflict. Section 6.6 states further constraints on how threads may be sequenced or interleaved.

6.5.1 Triggering Initialization

All Clock components contain an *initially* function which is used to initialize the component when it is first created. In request handlers, the *initially* function is used to establish an initial state for the request handler. In event handlers, the *initially* function establishes the state of local request handlers.

If a component has not yet been initialized, it must be initialized:

$$\forall c : component \,.\, (\neg \blacklozenge in(initTrigger\, c) \supset setFlag(initTrigger\, c)). \tag{6.46}$$

6.5.2 Dependencies among Components

In order to determine when the view and invariant functions of a component must be updated, we specify how components may depend on one another. Intuitively, a component's view function depends on a request handler if computation of the view depends on requests made to that request handler. This means that if the request handler's state is changed, the view function should be recomputed.

Then, in order to capture whether a view function of an event handler e depends on some request handler r, we write:

$$\begin{aligned}&viewDepends(e : eh, r : rh)\\ &\quad \equiv \exists t : thread \,.\, (\blacklozenge in(viewTrigger\, e, t) \wedge threadRequests(t, r)).\end{aligned} \tag{6.47}$$

That is, if the computation of a view has required this request handler in the past, the view *depends* on the request handler. Similarly, we define how an invariant may depend on a request handler:

$$\begin{aligned}&invDepends(e : eh, r : rh)\\ &\quad \equiv \exists t : thread \,.\, (\blacklozenge in(invTrigger\, e, t) \wedge threadRequests(t, r)).\end{aligned} \tag{6.48}$$

6.5.3 Triggering View and Invariant Generation

In order to specify how a component's view appears on the display, Clock programmers specify a view function indicating what the view looks like at any given time. The Clock system guarantees that whenever the view is out of date, it will be automatically updated. Similarly, whenever the state on which an invariant depends is modified, the invariant must be recomputed to ensure internal consistency within a component.

Recalling from figure 6.4 that the state of a request handler is represented as a variable, we can define that if a thread modifes state upon which the view and invariants of other components depend, view and invariant generation in those components must be triggered. The predicate *threadSetsFlag* is used to indicate that a thread t causes either the view or invariant flag of a component to be set.

Note in this definition that we specify only the conditions where *threadSetsFlag* must be true; an implementation can in fact choose to update views or invariants more often than necessary.

An invariant is triggered whenever state upon which the invariant depends is modified:

$$\begin{aligned}&\forall e : eh, t : thread, r : rh \,.\\ &\quad (threadUpdates(t, r) \wedge \neg threadFrom(t, invTrigger\, e) \wedge invDepends(e, r)\\ &\qquad \equiv threadSetsFlag(t, invTrigger\, e))\end{aligned} \tag{6.49}$$

Note that according to this definition, an invariant cannot trigger itself.

View updates are triggered whenever state upon which the view depends is modified:

$$\forall e : eh, t : thread, r : rh \,\boldsymbol{.}\, \\ (threadUpdates(t, r) \wedge viewDepends(e, r) \\ \equiv threadSetsFlag(t, viewTrigger\, e)) \tag{6.50}$$

Finally, we specify that if a thread t sets a flag f, that flag should be set:

$$\forall t : thread, f : flag \,\boldsymbol{.}\, (threadSetsFlag(t, f) \supset setFlag(f)). \tag{6.51}$$

6.6 Layer VI: Sequencing

The semantics of Clock specify that different threads may be executed concurrently – for example, the views of two components may be updated at the same time, or two user inputs coming from different input devices may be processed at the same time. Particularly in the case of multi-user implementations of Clock, this opportunity for concurrency becomes very important.

In Clock, concurrency is considered to be an implementation technique, rather than being explicitly programmed by the Clock programmer. Recognizing this, the Clock semantics provide an *illusion of single threadedness*, where the semantics guarantee that although threads may be executed concurrently for efficiency reasons, the programmer needn't be aware that the concurrency is present.

Intuitively, the illusion of single threadedness implies that if two inputs arrive in the system, the semantics must make it appear as if the input that arrived first is processed first. This means, for example, that two input threads may run concurrently; however, if the second thread modifies data that the first thread uses, the modification must be delayed until after the first thread has read whatever data it requires. This form of sequencing guarantees the same semantics as if the inputs were processed purely sequentially.

Additional sequencing constraints guarantee that if an input triggers the execution of invariants and updates of views, it must appear as if these updates take place before further inputs are processed.

The specification of sequencing of threads falls into two parts. An important predicate, *olderThread*, is defined. This predicate arranges threads into a partial order based on their priority. Intuitively, it must appear as if an older thread is executed before a newer thread. The second part specifies the illusion of single threadedness by specifying sequencing constraints on the activities of threads.

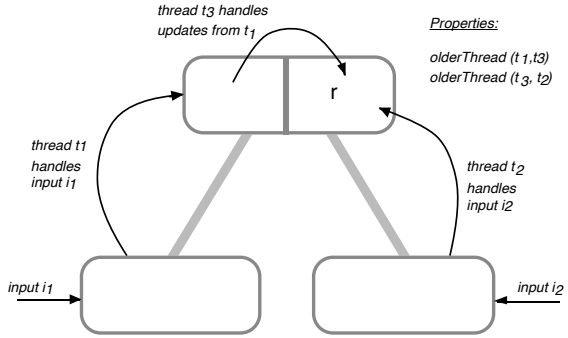


Figure 6.10: Example of the illusion of single-threadedness. If input i_1 arrives before input i_2, then the thread t_1 generated to handle i_1 has priority over thread t_2 generated to handle i_2. Assuming thread t_1 generates an update which is handled by thread t_3, then t_3 also has priority over t_2. This priority means that if t_1 or t_3 uses request handler r, then t_2 may have to block – e.g., if t_1 modifes r, then t_2 cannot read from r until the modification has taken place; if t_1 reads from r, then t_2 may not modify r until the read has taken place.

6.6.1 Ordering Threads

Threads in Clock are partially ordered over time. The predicate *olderThread* is defined to establish this ordering.

Two basic rules are used to establish the ordering:

- If the user performs two inputs, the one that was performed first is the one that is processed first;
- All consequences of an input are processed before the next input itself is processed.

This implies that the execution of programs must make it appear as if inputs are processed in order.

As seen in figure 6.10, when the processing of an input results in updates being sent up the tree, the threads to process these updates must also have priority over later user inputs. Similarly, invariant or view function updates resulting from an input take priority over later user inputs.

Linked Input Threads

We first define the notion of a *linked input*. In figure 6.10, the thread t_3 is executed as a result of thread t_1, and is therefore a linked input to t_1. The definition states that if one thread generates an update that is handled as an input in another thread, then the threads are linked:

$$\begin{aligned} &\mathit{linkedInput}(t_1, t_2 : \mathit{thread}) \\ &\quad \equiv \ \exists i : \mathit{uid}, e_1, e_2 : \mathit{eh} \,\boldsymbol{.}\, \\ &\qquad (\mathit{inThread}(i, t_1) \wedge \mathit{inThread}(i, t_2) \\ &\qquad\quad \wedge \mathit{outOccurs}(\mathit{updtOut}\, e_1, i) \wedge \mathit{inOccurs}(\mathit{updtIn}\, e_2, i)) \\ &\qquad \vee \exists t' \,\boldsymbol{.}\, (\mathit{linkedInput}(t_1, t') \wedge \mathit{linkedInput}(t', t_2)). \end{aligned} \tag{6.52}$$

Note that *linkedInput* is transitive and asymmetric.

Linked input threads follow the threads that caused them to be initiated:

$$\forall t_1, t_2 : \mathit{inputThread} \,\boldsymbol{.}\, (\mathit{linkedInput}(t_1, t_2) \supset \mathit{olderThread}(t_1, t_2)). \tag{6.53}$$

Linking Invariant and View Threads

If processing an input triggers an invariant thread, then the invariant thread is linked to the input. Similarly, if an invariant triggers further invariants lower down the tree, then these invariants are linked to the first one:

$$\begin{aligned} &\mathit{linkedInvThread}(t_1 : \mathit{thread}, t_2 : \mathit{invThread}) \\ &\quad \equiv \ \exists e : \mathit{eh} \,\boldsymbol{.}\, (\mathit{threadSetsFlag}(\mathit{invTrigger}\, e, t_1) \wedge \mathit{nextIn}(\mathit{invTrigger}\, e, t_2)) \\ &\qquad \vee \ \blacklozenge\, \mathit{linkedInvThread}(t_1, t_2) \vee \ \Diamond\, \mathit{linkedInvThread}(t_1, t_2). \end{aligned} \tag{6.54}$$

Similarly, if an input or invariant triggers a view update, the view update thread is linked to the thread that triggered it:

$$\begin{aligned} &\mathit{linkedViewThread}(t_1 : \mathit{thread}, t_2 : \mathit{viewThread}) \\ &\quad \equiv \ \exists e : \mathit{eh} \,\boldsymbol{.}\, (\mathit{threadSetsFlag}(\mathit{viewTrigger}\, e, t_1) \wedge \mathit{nextIn}(\mathit{viewTrigger}\, e, t_2)) \\ &\qquad \vee \ \blacklozenge\, \mathit{linkedViewThread}(t_1, t_2) \vee \ \Diamond\, \mathit{linkedViewThread}(t_1, t_2). \end{aligned} \tag{6.55}$$

Then, linked threads must follow the threads that triggered them:

$$\begin{aligned} &\forall t_1, t_2 : \mathit{thread} \,\boldsymbol{.}\, \\ &\quad ((\mathit{linkedInvThread}(t_1, t_2) \supset \mathit{olderThread}(t_1, t_2)) \\ &\qquad \wedge (\mathit{linkedViewThread}(t_1, t_2) \supset \mathit{olderThread}(t_1, t_2))). \end{aligned} \tag{6.56}$$

Two threads are then *linked* if there is an input, invariant or view link between them:

$$\begin{aligned} &\mathit{linkedThread}(t_1, t_2 : \mathit{thread}) \\ &\quad \equiv \ \mathit{linkedInput}(t_1, t_2) \vee \mathit{linkedInvThread}(t_1, t_2) \\ &\qquad \vee \mathit{linkedViewThread}(t_1, t_2). \end{aligned} \tag{6.57}$$

A thread may trigger a number of invariant threads or a number of view threads. When more than one thread is triggered, it is not set as to what order the threads are performed, but one has to take precedence over the other. We specify first the concept of a *main* thread:

$$\begin{aligned} &\mathit{mainThread}(t : \mathit{thread}) \\ &\quad \equiv \ (\mathit{inputThread}(t) \vee \mathit{invThread}(t) \vee \mathit{viewThread}(t)). \end{aligned} \tag{6.58}$$

Then, any pair of main threads must be ordered:

$$\forall t_1, t_2 : \mathit{mainThread} \,\boldsymbol{.}\, \mathit{olderThread}(t_1, t_2) \vee \mathit{olderThread}(t_2, t_1). \tag{6.59}$$

6.6.2 Sequencing User Input Threads

We distinguish those input threads which came as a result of user input. These threads have no predecessor threads that initiated them:

$$\begin{aligned} &\mathit{userInputThread}(t : \mathit{inputThread}) \\ &\quad \equiv \ \neg\exists t' : \mathit{inputThread} \,\boldsymbol{.}\, \mathit{linkedInput}(t', t). \end{aligned} \tag{6.60}$$

We then specify that input threads are ordered according to the inputs that generate them. That is, if the user performs two actions in sequence, then the threads handling the actions should be ordered in the same sequence. We specify that if input i_1 was performed by the user before input i_2, then thread t_1 handling i_1 should come before thread t_2 handling i_2, and that all input, invariant, and view threads linked to t_1 should also come before t_2:

$$\begin{aligned} &\forall t_1, t_2 : \mathit{userInputThread}, i_1, i_2 : \mathit{uid}, e_1, e_2 : \mathit{eh} \,\boldsymbol{.}\, \\ &\quad \mathit{pending}(\mathit{updtIn}\, e_1, i_1) \wedge \mathit{nextIn}(\mathit{eventId}\, e_1, t_1) \\ &\qquad \wedge \mathit{pending}(\mathit{updtIn}\, e_2, i_2) \wedge \mathit{nextIn}(\mathit{eventId}\, e_2, t_2) \\ &\qquad\quad \wedge (\mathit{out}(i_1) \blacktriangleright \mathit{out}(i_2)) \\ &\qquad\quad\quad \supset \mathit{olderThread}(t_1, t_2) \\ &\qquad\quad\quad\quad \wedge \forall t' : \mathit{thread} \,\boldsymbol{.}\, (\mathit{linkedThread}(t_1, t') \supset \mathit{olderThread}(t', t_2)). \end{aligned} \tag{6.61}$$

6.6.3 Illusion of Single-Threadedness

As seen in figure 6.10, different threads may contend for the same resources. The illusion of single-threadedness states that as long as two threads do not require the same resources, the order of their execution is immaterial. However, when the threads compete for a resource, the priority of the threads (as determined by *olderThread*) determines the order of access to the resource.

If two threads t_1 and t_2 both wish to use a request handler r, and t_1 is the older thread, then the illusion of single-threadedness specifies that no action on the part of t_2 may modify the data seen by r. Further, if t_1 modifies r, then t_2 may not access r before

the modifications take place, guaranteeing that the execution is as if t_1 had completely executed before t_2.

We then specify the illusion of single-threadedness, which says that if thread t_1 is older than thread t_2, then it must appear as if t_1 executes to completion before t_2 executes. In particular, t_1 has priority access to shared data over t_2:

$$\begin{aligned}\forall t_1, t_2 : \mathit{thread}, r : \mathit{rh} \,.\quad & (6.62)\\ \mathit{olderThread}(t_1, t_2)\\ \supset\ & (\ \Diamond\, \mathit{threadRequests}(t_1, r) \supset \neg \mathit{threadUpdates}(t_2, r))\\ \wedge\ & (\ \Diamond\, \mathit{threadUpdates}(t_1, r)\\ & \quad \supset \neg(\mathit{threadRequests}(t_2, r) \vee \mathit{threadUpdates}(t_2, r)))\\ \wedge\ & (\ \Diamond\, \mathit{threadViews}(t_1, v) \supset \neg \mathit{modifiesVar}(t_2, v)).\end{aligned}$$

Finally, within a thread itself, all updates to a request handler made by the thread are deferred until all requests to that request handler have taken place:

$$\begin{aligned}\forall t : \mathit{thread}, r : \mathit{rh} \,.\quad & (6.63)\\ \Diamond\, \mathit{threadRequests}(t, r) \supset \neg \mathit{threadUpdates}(t, r).\end{aligned}$$

6.7 Conclusion

This chapter has presented the semantics of Clock, specified using the TCFP framework. The semantics provide a precise definition of Clock, and at the same time gives us an opportunity to evaluate the effectiveness of TCFP in describing the semantics of programming languages.

One of the primary advantages of TCFP is that it is possible to build semantics in layers, based on existing abstractions. As was seen in figures 6.2 and 6.4, event and request handlers are modeled using the various structures defined in chapter 5. These structures are not specific to Clock itself, and could be used in the semantics of any language.

Once the basic structure of the components is defined, the definitions specific to the Clock language itself are built in a series of six layers. Each layer builds on the definitions of the layers below it. Therefore, a good separation of concerns is provided within the semantics, allowing a reader of the semantics to look at various features of the definition in isolation. For example, to understand how invariant and view functions are triggered, it is not necessary to understand the concurrency issues of the language. This separation of concerns is a major benefit in a language as complex as Clock.

The handling of concurrency and non-determinism shows the true advantages of TCFP. It is quite simple to specify that view and invariant functions can be executed at any time, but that certain concurrency constraints must be observed to provide the illusion of single-threadedness. In this sense, the semantics of the language lend themselves naturally to the generate and restrict approach of TCFP.

Over all, the description of the semantics is lengthy and detailed. Given the complex nature of the Clock language, particularly involving the features of concurrent execution, this is not surprising. What is hoped is that through the clear, layered structure of the semantics, and through the separation of concerns, the description is still reasonably accessible. The fact that it was possible to describe the semantics of Clock at all represents an advance, in that formal methods have not gained wide acceptance in describing the semantics of user interface languages.

Based on the semantics presented here, chapter 7 will present some of the interesting formal properties of Clock.

Chapter 7

Properties of Clock

The semantics of chapter 6 gives us a sufficient framework to investigate some of the key properties of the Clock language. There are many properties which we might choose to examine; however, we shall concentrate on the declarative properties discussed in Chapter 4. This chapter proposed that languages such as Clock should consider *declarativeness in the small*, corresponding to such traditional properties as referential transparency, and *declarativeness in the large*, examining the properties of high-level I/O and constraint systems.

In this chapter, we prove two theorems, showing that Clock does provide properties of declarativeness in the small and in the large. The first (theorem 7.1.1) specifies that all functions written by Clock programmers are referentially transparent; the second (theorem 7.2.2) shows that all sequences of triggered invariant and view function evaluations are finite. Together, these theorems show that the implementation model of Clock is not revealed to Clock programmers, thus providing the most basic property of declarative programming. In more detail, these properties are:

Referential Transparency: The Clock language itself clearly provides imperative features, such as the ability to manipulate global state through request handlers, and the ability to post views to the display. In Clock, however, these imperative features are specified on the level of the architecture. The actual functions that users write are themselves pure functions, similar to thos of languages such as Haskell [56] or LML [5]. We consider functions to be pure if they are *referentially transparent*; that is, within a computation, a function applied to a given argument always has the same value.

The theorem proving that all user-defined functions are referentially transparent (theorem 7.1.1) establishes the main success of Clock's design – that programmers can write purely functional code to specify user interfaces, without having to worry about the impure effects of reading user inputs or handling display updates.

Termination of Constraint Sequences: As was seen in chapter 6, the execution of a single user input can lead to the triggering of a sequence of invariant and view threads intended to bring all request handlers and views up to date. This

automatic triggering of invariants and views is similar to the *constraint* facilities of languages such as Garnet [74] and RendezVous [49]. As was seen in chapter 2, in these languages, sequences of constraints triggering other constraints are not guaranteed to terminate. We call such non-terminating sequences *infinite constraint sequences*.

In Clock, the architecture structure of the language guarantees that any sequence of invariant and view updates resulting from an input is guaranteed to terminate. This means that as long as the individual invariant and view functions themselves terminate, the sequence of updates will also eventually terminate (theorem 7.2.2). A corollary to this theorem is that any input a user makes will eventually be processed.

This property of termination is critical to the success of Clock's high-level programming model. If it were possible to generate looping constraint structures, programmers would have to know exactly when invariant and view functions would be triggered in order to avoid programming infinite sequences, or to debug them when they occur. Since infinite constraint sequences are impossible to program, programmers do not need to be aware of the execution order used by a particular implementation.

This chapter investigates these properties of the Clock language, using the semantic description of Clock to justify them. Ultimately, the proof of these properties are proofs in interaction logic, using the proof system defined in appendix B. The proofs are based on the constraints defined in the semantics of chapter 6. Since these proofs quickly become long and detailed, our presentation intermixes proof sketches with fully rigorous proofs.

7.1 Referential Transparency

Referential transparency is one of the key defining properties of pure functional languages. Intuitively, this property means that the value of an expression does not depend on its context, or on the execution order of the program. For example, if an expression e is referentially transparent, then in the expression:

$$e + e$$

both occurrences of e evaluate to the same value, and the expression is equivalent to:

$$2 * e$$

In appendix A, the property of referential transparency is investigated in terms of the $I\lambda c$ language. A precise definition of referential transparency is given (definition A.3.1), stating that an expression is referentially transparent iff it evaluates to the same value, regardless of the context in which it is evaluated. As discussed in appendix A, the basic core of $I\lambda c$ is referentially transparent (theorem A.3.2). This is intuitively to

be expected, since Basic $I\lambda c$ is only a slightly extended version of the λ-calculus, which possesses the referential transparency property. Full $I\lambda c$, however, possesses I/O constructs that are not referentially transparent. For example, the expression

```
read "stdin"
```

takes on different values depending on the input provided by the user.

Programs in the Clock language are constructed using a graphical architecture language to specify components and connection of components, and using a syntactically sugared form of $I\lambda c$ to program the components themselves. It is therefore not immediately clear that expressions written by Clock programmers will be always be referentially transparent.

In fact, as was claimed in chapter 4, any function written by a Clock programmer is referentially transparent. The argument why this is true is as follows:

The execution of a Clock program is not a single computation, but rather a *set* of computations, partially ordered by time. Each computation corresponds to the execution of one of the functions programmed by a Clock programmer; in chapter 6, each execution is represented by a *main* thread, partially ordered by the *olderThread* predicate. Each of these functions is guaranteed to be executed so that its referential transparency is preserved. This means, for example, that if a view function makes the same request twice during its execution, then the response to both requests will be the same. During two separate executions of a view function, however, the request may return a different value.

The functions provided by the Clock programmer are programmed using Basic $I\lambda c$, extended with two constructs: *requests* can be used to obtain values from request handlers further up the tree, and the values of *subview variables* can be referenced when computing a view. Since the Basic $I\lambda c$ constructs are known to be referentially transparent (theorem A.3.2), it only remains to be shown that requests and subview references are referentially transparent within the evaluation of a main thread. This property follows directly from definitions 6.62 and 6.63, which require that no thread modify the state of request handlers or subview variables until the current thread has finished using them.

This referential transparency property means that Clock programmers can have available the facilities of persistent state and graphical I/O, which still enjoying the usual mathematical properties of functional programming.

7.1.1 Proof of Referential Transparency

The final proof of referential transparency in Clock is built up from a series of smaller results. The first of these is that the response values generated by request handlers depend on only the request itself, and the request handler's state. That is, as long as the state of a request handler remains constant, then the response to each request rq will always be the same. We first define a predicate *responseVal* capturing what the

response is to a given request under a given state:

$$\begin{aligned} &responseVal(r : rh, rq : request, resp : value, s : value) \qquad (7.1)\\ &\quad \equiv \ \exists t : thread, i_1, i_2, i_3 : uid \,.\\ &\qquad threadFrom(t, rqId\, r)\\ &\qquad \wedge\, inThread(i_1, t) \wedge inThread(i_2, t) \wedge inThread(i_3, t)\\ &\qquad \wedge\, inOccurs(reqIn\, r, rq, i_1)\\ &\qquad \wedge\, inOccurs(state\, r, s, i_2)\\ &\qquad \wedge\, outOccurs(respOut\, r, resp, i_3). \end{aligned}$$

We can then prove that responses change only if the request handler's state changes. That is, if the same request is sent twice to the same request handler, with the same state, then the response is guaranteed to be the same:

Lemma 7.1.1 (Referential Transparency of Request Handlers)

$$\begin{aligned} &\forall r : rh, rq : request, resp, resp' : value, s : value \,.\\ &\qquad responseVal(r, rq, resp, s) \wedge responseVal(r, rq, resp', s) \supset resp = resp'. \end{aligned}$$

Proof: If "$responseVal(r, rq, resp, s)$" then it holds that "$resp =$ `requestFn s rq`" (definition of request handlers, figure 6.5), where "`requestFn`" is the function handling requests for r. Similarly, if "$responseVal(r, rq, resp', s)$", then "$resp' =$ `requestFn s rq`". By the definition of the Clock language, "`requestFn`" is constructed from Basic $I\lambda c$ only; therefore, by theorem A.3.2, "`requestFn s rq`" is referentially transparent, and has a unique value. Therefore, "$resp = resp'$".

□

Ultimately, we are interested in showing that functions executed within a main thread are referentially transparent. One required property is that requests return the same value within the execution of a main thread. To investigate this property, we define a predicate *threadRqResp* that holds if during the execution of a main thread, a request rq is made, and a response of $resp$ is returned. Requests and responses are identified by their unique identifiers, so the predicate holds if a request rq with uid $\subset_{rq}$ is sent, and a response $resp$ with uid i_{resp} is returned, and both of the unique identifiers ($\subset_{rq}$ and i_{resp}) belong to the same thread t. This definition will eventually be used to show that *all* requests made during the execution of a main thread return the same value.

$$\begin{aligned} &threadRqResp(t : mainThread, rq : request, i_{rq} : uid, resp : value, i_{resp} : uid) \qquad (7.2)\\ &\quad \equiv \ inThread(i_{rq}, t) \wedge inThread(i_{resp}, t)\\ &\qquad \wedge\, outOccurs(rq, i_{rq}) \wedge inOccurs(resp, i_{resp})\\ &\qquad \wedge\, respondsTo(i_{resp}, i_{rq}). \end{aligned}$$

We next establish that all requests generated within a single main thread must be generated by the same event handler. Intuitively, this holds because main threads are generated from *group* constructs in the definition of an event handler (figure 6.3), where the grouped code does not cross event handler boundaries. This lemma will allow us to determine which request handlers handle requests generated by a thread, by linking each main thread to a specific event handler. The lemma states that each request issued by a main thread is written to the "*reqOut*" port of the same event handler e.

Lemma 7.1.2 (Threads bound to a single EH)

$$\begin{aligned} &\forall t : mainThread, rq : request \,.\\ &\qquad \exists e : eh \,.\, \forall rq : request, i_{rq} : uid \,.\\ &\qquad\quad threadRqResp(t, rq, i_{rq}) \supset outOccurs(reqOut\, e, rq, i_{rq}). \end{aligned}$$

Proof: Let $t : mainThread$. Then by definitions 6.58 and 6.26-6.29, it holds that $\exists o : oracle, e : eh \,.\, inOccurs(o, t) \wedge (o = eventIde \vee o = invTriggere \vee o = viewTriggere)$. By definition 5.33, the thread is unique; i.e., $\forall o' : oracle \,.\, inOccurs(o', t) \supset o' = o$. Then by definition 6.23 and the definition of event handlers (figure 6.3), $\forall e' : eh, i : uid \,.\, outOccurs(reqOut\, e, i) \wedge inThread(i, t) \supset e' = e$, and lemma follows.

□

From layer VI of the Clock semantics, we saw that I/O is ordered so that a thread may not modify the state of request handlers upon which earlier threads rely. A simple consequence of this definition is that the state of a request handler does not change throughout a thread. In particular, if a thread t makes a request to request handler r, then during any future requests to r within thread t, r will have the same state:

Lemma 7.1.3 (RH State Constant During Thread)

$$\begin{aligned} &\forall t : mainThread, r : rh, s : value \,.\\ &\qquad (threadRequests(t, r) \wedge val(state\, r, s)\\ &\qquad\quad \supset (threadRequests(t, r) \sqsupset\ val(state\, r, s))). \end{aligned}$$

Proof: For $t : mainThread$, $r : rh$, the sequencing constraints 6.62 and 6.63 show that $threadRequests(t, r) \supset \forall t' \,.\, threadRequests(t, r) \lhd threadUpdates(t', r)$; i.e., if a request is made to a request handler, all updates are deferred until all further requests are made. By wire definition 5.17, no update can be read by r if no updates are sent to r: $threadRequests(t, r) \lhd in(updtIn\, r)$. By the definition of request handlers (figure 6.5), the state can only be modified if an update is processed: $threadRequests(t, r) \lhd out(state\, r)$. By the definition of variables (definition 5.29), variables change only when written to; therefore, lemma holds.

□

From lemmas 7.1.2 and 7.1.3, we can prove that whenever a main thread makes the same request multiple times, the response to the request is the same. This property shows that requests are referentially transparent within the execution of a thread.

Lemma 7.1.4 (Referential Transparency of Requests)

$$\forall t : mainThread, rq : request, resp, resp' : value \,.\, (threadRqResp(t, rq, resp) \wedge threadRqResp(t, rq, resp') \supset (resp' = resp)).$$

Proof: Assume there exist i_{rq}, i_{resp}, i'_{rq}, i'_{resp} : *uid*, t : *mainThread*, rq:*request*, $resp$,$resp'$: *value* s.t. $threadRqResp(t, rq, i_{rq}, resp, i_{resp})$ and $threadRqResp(t, rq, i'_{rq}, resp', i'_{resp})$. Then by lemma 7.1.2, there exists e : *eh* s.t. $outOccurs(reqOut\, e, rq, i'_{rq}) \wedge outOccurs(reqOut\, e, rq, i'_{rq})$. By the definition of *threadRqResp* (definition 7.2), $respondsTo(i_{resp}, i_{rq}) \wedge respondsTo(i'_{resp}, i'_{rq})$. From the definitions of routing (definitions 6.34 and 6.15), it holds that both requests are handled by the same request handler: there exists r : *rh* s.t. $inOccurs(reqIn\, r, i_{rq}) \wedge inOccurs(reqIn\, r, i'_{rq})$. Then, by lemma 7.1.3, if $in(reqIn\, r, i_{rq}) \sqsupseteq val(state\, r, s)$ then $in(reqIn\, r, i'_{rq}) \sqsupseteq val(state\, r, s)$. Then by lemma 7.1.1, $resp = resp'$.

□

We can now show that any function written by a Clock programmer is referentially transparent. We have to show that any expression evaluated within a main thread has the same value regardless of when it is evaluated within the thread. To prove this, we must show that requests and subview references always have the same values within the execution of the thread, which follows directly from the lemmas just proved.

Theorem 7.1.1 (Referential Transparency of Clock Functions)

Every Clock expression e is referentially transparent within the execution of a main thread.

Proof: e can appear within a request handler or within an event handler. If e appears in a request handler, then by syntactic restrictions in Clock, e contains only constructs from Basic $I\lambda c$, and by theorem A.3.2, e has a unique value. If e appears in an event handler, then by definition of event handlers (figure 6.3), e occurs within a main thread. Ie, $\exists t$: *mainThread* s.t. $\forall i$: *uid* . $ioOccurs(i)$ and i was generated by the execution of e implies that $inThread(i, t)$. By definition A.3.1, e is referentially transparent iff e has a unique value under all evaluation contexts. Using structural induction over expressions to establish this property, we have base cases: if e in Basic $I\lambda c$, e has a unique value by theorem A.3.2. If e is a request, e has a unique value by lemma 7.1.4. If e is a subview reference, e has a unique value, by definition 6.63. From the base case, the inductive step follows in a straight-forward manner.

□

7.2 Termination Properties

The last section demonstrated that functions written by Clock programmers are guaranteed to be referentially transparent. Clock programs, however, consist of sequences of functional computations, as guided by the structure of the program's architecture.

In this section, we wish to demonstrate that Clock's declarative properties extend also to the architecture level. In Clock, whenever the user makes an input, an input thread is invoked to handle the input. This input thread may in turn invoke other input threads, invariant threads, and view threads. In chapter 4, we claimed that in all Clock programs, this sequence of threads invoking new threads is guaranteed to terminate – that is, it is impossible to write the equivalent of infinite constraint sequences such as those found in Garnet (see chapter 2).

What this section will show is that as long as the individual functions used to implement views, invariants and inputs themselves terminate, then the sequence of threads used to perform an update will also terminate. This property is called *weak termination*, to emphasize that termination is guaranteed as long as the individual functions terminate.

The presence of weak termination is reassuring. It means that programmers cannot unintentionally introduce unresolvable dependencies among components. Because of this, programmers are never faced with debugging constraint loops or infinite constraint sequences, and therefore do not need to understand the evaluation strategy used in processing user inputs. Weak termination guarantees a form of declarativeness in Clock architectures, since programmers never need to know when or in what order component functions are executed in order to understand a program.

In order to establish weak termination, we demonstrate a sequence of smaller properties. First we show that any sequence of *input threads* generated by a user input terminates; that is, that there is always a *last* input thread in any sequence. Secondly, we show that any sequence of invariants terminates. From the fact that view threads cannot trigger other threads, we can then show that any user input generates a finite sequence of threads.

Before establishing these properties, we first develop two important proof techniques used for reasoning about architecture trees.

7.2.1 Architecture Induction

In most branches of mathematics, some form of inductive proof technique is developed to aid in proving theorems applying to all elements of a structure. In number theory, mathematical induction is used, while in programming language semantics, structural induction is a key proof technique. Similarly, for reasoning about Clock architectures, we develop two variants of *architecture induction*.

Assume we wish to prove that some property p holds over all elements of an architecture; i.e., that $\forall e$: *eh*. $p(e)$. Using architecture induction, we would be obliged to prove that p holds at the root of the tree, and that if p holds at some element e_p : *eh*, then p

holds over all children of e_p. From this, we can conclude that p holds over all elements of the tree.

Intuitively, this property holds, since if the property holds at the root, it also holds for all children of the root, and all children of the children of the root, and so forth until the entire tree is covered. More formally, the property can be proved as a theorem:

Theorem 7.2.1 (Architecture Induction)

For all unary predicates p, it holds that:

$$\begin{aligned}&\forall e : eh \,.\, (root(e) \supset p(e)) \\ &\qquad \wedge \forall e_p, e_c : eh \,.\, (ehChildOfEh(e_c, e_p) \wedge p(e_p) \supset p(e_c)) \\ &\quad \supset \forall e : eh \,.\, p(e).\end{aligned}$$

Proof: Assume (i) $\forall e : eh.\ (root(e) \supset p(e))$ and (ii) $\forall e_p, e_c : eh.\ (ehChildOfEh(e_c, e_p) \wedge p(e_p) \supset p(e_c))$, and let $e : eh$. Then by definitions 6.17 and 6.18, $inTree(e)$ and $root(e) \vee \exists e_p : eh \,.\ \ ehChildOfEh(e, e_p) \ \wedge inTree(e_p)$. Case 1: if $root(e)$, then $p(e)$, by (i). Case 2: if $\exists e_p : eh \,.\, ehChildOfEh(e, e_p) \wedge inTree(e_p)$, then for some $e_1, \ldots, e_n : eh$, we can derive a set of sentences $root(e_1)$, $ehChildOfEh(e_1, e_2)$, …, $ehChildOfEh(e_{n-1}, e_n)$, and $ehChildOfEh(e_n, e)$, by successively applying definition 6.17. By (i), $p(e_1)$. Then by successive application of (ii), we derive that $p(e_2)$, …, $p(e_n)$, and hence $p(e)$.

□

A variant on architecture induction is called *course-of-values* architecture induction. For some property p and some given event handler e_c, if assuming p holds for all event handlers above e_c allows us to prove that p holds for e_c also, then we can infer that $\forall e : eh \,.\, p(e)$. COV architecture induction is a straightforward corollary of theorem 7.2.1.

Corollary 7.2.1 (Course of Values Architecture Induction)

For all unary predicates p, it holds that:

$$\begin{aligned}&(\forall e_c : eh \,. \\ &\quad \forall e_p : eh \,.\, above(e_p, e_c) \supset p(e_p)) \supset p(e_c)) \\ &\qquad \supset \forall e : eh \,.\, p(e).\end{aligned}$$

Proof: Assume $(\forall e_c : eh \,.\, \forall e_p : eh \,.\, above(e_p, e_c) \supset p(e_p)) \supset p(e_c))$. Step 1: Assume $e_r : eh$ and $root(e_r)$. Show $p(e_r)$: It holds that $(\forall e_p : eh \,.\, above(e_p, e_r) \supset p(e_p)) \supset p(e_r))$. But $\neg\exists e_p \,.\, above(e_p, e_r)$ (definition 6.32). Therefore $p(r)$ holds. Step 2: Let $e_p, e_c : eh$, where $ehChildOfEh(e_c, e_p)$, and let $p(e_p)$. Show $p(e_c)$: Since $ehChildOfEh(e_c, e_p)$, it follows that $above(e_p, e_c)$, from which we have $p(e_c)$.

□

7.2.2 Termination of Input Sequences

Our next step in establishing the *weak termination* property is to show that sequences of input threads eventually terminate.

When a Clock program takes a user input, the input is directed towards a component e, and an event function in e is executed to handle the input. Executing the event function may cause updates to be sent up the tree, each of which will be handled as a linked input thread.

The proof that such a sequence of inputs terminates is conceptually simple – first we demonstrate that all updates are sent up the tree. Then, since updates must eventually reach the root, there must eventually be a last one.

First, we demonstrate that linked input threads move up the tree. The proof is a straight-forward application of COV architecture induction:

Lemma 7.2.1 (Linked Updates Move Up Tree)

$$\begin{aligned}&\forall t_1, t_2 : inputThread, e_1, e_2 : eh \,. \\ &\quad linkedInput(t_1, t_2) \\ &\qquad \wedge threadFrom(eventId\, e_1, t_1) \wedge threadFrom(eventId\, e_2, t_2) \\ &\qquad\quad \supset above(e_2, e_1).\end{aligned}$$

Proof: (By architectural induction.) Define

$$\begin{aligned}&p(e_1 : eh) \ \equiv \\ &\quad \forall t_1, t_2 : inputThread, e_2 : eh \,. \\ &\qquad linkedInput(t_1, t_2) \\ &\qquad\quad \wedge threadFrom(eventId\, e_1, t_1) \wedge threadFrom(eventId\, e_2, t_2) \\ &\qquad\qquad \supset above(e_2, e_1).\end{aligned}$$

Assume $e_c : eh$, and $\forall e_p : eh \,.\, (above(e_p, e_c) \supset p(e_p))$. Show $p(e_c)$: Assume $linkedInput(t_1, t_2) \wedge threadFrom(eventId\, e_1, t_1) \wedge threadFrom(eventId\, e_2, t_2)$. From the definition of *linkedInput* (definition 6.52), there are two cases. Case 1: $\exists i : uid, u : update$ s.t. $inThread(i, t_1) \wedge inThread(i, t_2) \wedge outOccurs(updtOut\, e_1, u, i) \wedge inOccurs(updtIn\ e_2, u, i)$. By definitions 5.16 and 5.14, $out(updtOut\ e_1, i) \supset connected(updtOut\ e_1, updtOut\ e_2)$. Therefore, by definitions 6.37 and 5.15, $handlesUpdate(e_2, u, e_1)$. Therefore, by definitions 6.36 and 6.33, $above(e_2, e_1)$, establishing $p(e_c)$. Case 2: $\exists t' \,.\, linkedInput(t_1, t') \wedge linkedInput(t', t_2)$. By case 1, there exists $e' : eh$ s.t. $threadFrom(eventIde', t') \wedge above(e', e_1)$; since $above(e;, e_1)$, it holds that $p(e')$, giving $above(e_2, e')$. By definition 6.32, it therefore holds that $above(e_2, e_1)$, establishing $p(e_c)$.

□

We now define a predicate *noLinkedInput* to indicate whether an input thread is the last one in a sequence of linked inputs:

$$noLinkedInput(t : inputThread) \equiv \neg\exists t' : inputThread \,.\, linkedInput(t, t'). \quad (7.3)$$

A sequence of linked inputs terminates iff there is a "last" thread in the sequence; that is, if there is some thread t' in the sequence such that $noLinkedInput(t')$. From lemma 7.2.1 and using COV architecture induction, we can demonstrate that all sequences of linked inputs terminate:

Lemma 7.2.2 (Termination of Update Sequences)

$$\forall t : inputThread \,.\, noLinkedInput(t) \\ \vee \exists t' : inputThread \,.\, linkedInput(t, t') \wedge noLinkedInput(t').$$

Proof: (By COV architecture induction.) Define:

$$p(e_1 : eh) \equiv \\ \forall t_1 : inputThread \,. \\ threadFrom(eventId\, e_1, t_1) \\ \supset (noLinkedInput(t_1) \\ \vee \exists t' : inputThread \,.\, linkedInput(t, t') \wedge noLinkedInput(t')).$$

We wish to show $\forall e : eh \,.\, p(e)$. Let $e_1 : eh$. Induction hyp: assume $\forall e_2 : eh \,.\, above(e_2, e_1) \supset p(e_2)$. Induction step: show $p(e_1)$. Let $t_1 : inputThread$, where $threadFrom(eventId e_1, t_1)$. Then it holds that (i) $noLinkedInput(t_1)$, or (ii) $\neg noLinkedInput(t_1)$. From (i), $p(e_1)$ follows directly. From (ii), we must show:

$$\exists t' : inputThread \,.\, linkedInput(t_1, t') \wedge noLinkedInput(t').$$

By definition 7.3,

$$\exists t_2 : inputThread \,.\, linkedInput(t_1, t_2).$$

Let $e_2 : eh$ s.t. $threadFrom(eventId\, e_2, t_2)$. By lemma 7.2.1, $above(e_2, e_1)$, and therefore, by induction hyp, $p(e_2)$. Then by definition of p, either (a) $noLinkedInput(t_2)$, from which $p(e_1)$ follows directly, or (b) there exists t' : *linkedInput* satisfying $linkedInput(t_2, t') \wedge noLinkedInput(t')$. Then by definition 6.52, $linkedInput(t_1, t')$, which implies $p(e_1)$. Therefore, by corollary 7.2.1, $\forall e : eh \,.\, p(e)$, from which the lemma follows.

□

7.2.3 Termination of Invariant and View Sequences

Updates may also trigger invariants, as state changes caused by updates cause components to become inconsistent. Invariants also cause state change, and therefore may themselves trigger invariants. We now show that all sequences of invariants triggered by an update are finite.

The basis of this proof is the following: invariant functions can make requests upwards in the tree. Therefore, any invariant must be triggered by changes occurring at some higher node in the tree. Invariants also may only modify local state. Therefore, the execution of the invariant of some event handler e can only trigger the invariants of components below e in the tree. Since trees have finite depth, this means that every sequence of invariants must eventually terminate.

First, we show that if an invariant triggers another invariant, the triggered invariant must belong to an event handler lower down in the tree:

Lemma 7.2.3 (Invariants Move Down Tree)

$$\forall t_1, t_2 : invThread, e_1, e_2 : eh \,. \\ linkedInvThread(t_1, t_2) \\ \wedge threadFrom(eventId\, e_1, t_1) \wedge threadFrom(eventId\, e_2, t_2) \\ \supset above(e_1, e_2).$$

Proof: Let $t_1, t_2 : invThread$ and let $e_1, e_2 : eh$. Let $linkedInvThread\ (t_1, t_2)$ and $threadFrom\ (eventId\, e_1, t_1)$ and $threadFrom\ (eventId\, e_2, t_2)$. Show $above\ (e_1, e_2)$. $linkedInvThread\ (t_1, t_2)$ implies that at some time, $threadSetsFlag\ (invTrigger\, e_2, t_1) \wedge nextIn(invTrigger\ e_2, t_2)$ (def of *linkedInvThread*; def 6.54). Then for some $r : rh$, $threadUpdates\ (t_1, r) \wedge invDepends(e_2, r) \wedge \neg threadFrom(t_2, invTrigger\, e_1)$ (def 6.49). From $threadUpdates(t_1, r)$, we know that $rhBelongsToEh(r, e_1)$ (def 6.45), and from $invDepends(e_2, r)$, we know that $above(r, e_2)$ (defs 6.48 and 6.36). Since $threadFrom(t_2, invTriggere_2)$ and $\neg threadFrom(t_2, invTriggere_1)$, it holds that $e_1 \neq e_2$. Since $above(r, e_2)$ and $rhBelongsToEh(r, e_1)$, then by def 6.32, it holds that $above(e_2, e_1)$.

□

We define that a thread is *finished* if there will be no more I/O activity within the thread:

$$finishedThread(t : thread) \equiv \Diamond(\forall i : uid \,.\, io(i) \supset \neg inThread(i, t)). \quad (7.4)$$

We then state that for all sequences of invariants, eventually the sequence terminates. Ie, eventually, all linked invariants are finished:

Lemma 7.2.4 (Termination of Invariant Sequences)

$$\forall t : \mathit{invThread} \,.\, \Diamond\,(\forall t' : \mathit{invThread} \,.\, \mathit{linkedInvThread}(t, t') \supset \mathit{finishedThread}(t')).$$

Proof: By COV architecture induction, similarly to lemma 7.2.2.

□

Input and invariant threads can trigger view threads. View threads may not perform updates, however, and therefore themselves generate no linked threads:

Lemma 7.2.5 (Termination of View Sequences)

$$\forall t : \mathit{viewThread} \,.\, \neg\exists t' : \mathit{thread} \,.\, \mathit{linkedThread}(t, t').$$

Proof: Follows directly from definition of *linkedThread* (def. 6.57).

□

7.2.4 Proof of Limited Termination

Having investigated how input and invariant threads can generate linked threads, and having shown that these sequences of linked threads always terminate, we can finally express our notion of limited termination within Clock programs. *Limited termination* means that a user input may cause an arbitrarily long sequence of linked inputs, invariants and view updates to be triggered. However, this sequence is always guaranteed to finish eventually. In Clock, as in all programming languages, it is possible to write programs with errors in them. Theorem 7.2.2 shows, however, that it is impossible to write a program where an infinite sequence of linked inputs, invariants or views results. The theorem is as follows:

Theorem 7.2.2 (Termination)

$$\forall t : \mathit{userInputThread} \,.\, \Diamond\,(\forall t' : \mathit{thread} \,.\, \mathit{linkedThread}(t, t') \supset \mathit{finishedThread}(t')).$$

Proof: Follows from the termination of linked update sequences (lemma 7.2.2), linked invariant sequences (lemma 7.2.4) and view sequences (lemma 7.2.5).

□

7.3 Conclusion

This chapter has demonstrated two important properties of the Clock language. The *referential transparency* property (theorem 7.1.1) showed that any expression written by a Clock programmer is referentially transparent. Referential transparency is one of the key defining properties of functional languages. This means that despite the imperative extensions made to Clock, programmers may act as if they are programming in a pure functional language.

After a user of a Clock program performs some input, a sequence of linked inputs may be generated, as well as a set of invariant and view functions. The *limited termination* property (theorem 7.2.2) showed that the number of such triggered threads is always finite. This implies that a Clock programmer need not worry about the possibility of writing infinite constraint sequences, where invariant and input functions can have unresolvable dependencies.

These two properties can both be seen as contributing towards the declarative nature of Clock. Referential transparency represents declarativeness in the small – i.e., the evaluation of individual expressions is not dependent on timing or side-effects. Limited termination represents declarativeness in the large – programmers can write functions handling inputs, invariants and views without having to worry about the order in which they are evaluated, or the possible interdependencies the functions may have.

The proofs of these two properties provide examples of how it is possible to reason about a language within the TCFP framework. Once the semantics of a language have been defined in terms of TCFP constraints (as was done in chapter 6), language properties can be proved as consequences of the semantics. Proving language properties serves the dual purpose of gaining better understanding of the language, and of testing the correctness and completeness of the semantics.

Proofs in TCFP are not trivial. The description of a language such as Clock is long and complicated, and logical proofs are often lengthy and arduous. Without some form of mechanical proof aid, it is challenging to prove all properties of a language in full detail. An interesting future research topic would be to develop mechanical support for reasoning within TCFP.

Chapter 8

Conclusion

The development of graphical, direct manipulation user interfaces introduces challenges not present in the development of traditional interactive applications. Graphical user interfaces must be developed by iterative refinement, requiring good support for rapid prototyping and easy program modification. Rapid prototyping is difficult, however, when programmers need to deal with inputs occurring in non-deterministic order, concurrency, consistency maintenance, and semantic feedback. Providing programming language support for user interface development is therefore a difficult task.

The Clock language was designed to support rapid prototyping while providing strong support for program structuring, easy program modification, non-determinism, and concurrency. The key to Clock's design is the provision of a high-level programming model. This model provides flexible support for user interface development, combining an architecture language for structuring user interfaces, constraint-style display and consistency maintenance, and a simple, high-level I/O system. Clock is purely declarative, both in the small and in the large.

In summarizing the results of this thesis, we first consider the importance of declarative programming, and how Clock supports it. We then discuss the relationship between Clock and the TCFP framework upon which Clock is based; following this, we discuss the success of TCFP framework itself. The chapter concludes with a summary of future and ongoing work with the Clock language.

8.1 Clock and Declarative Programming

As we saw in chapter 2, the specificational flavour of declarative programming is particularly suited to programming user interfaces. In developing user interfaces, programmers must keep in mind:

- What inputs might the user perform next?
- How will these inputs affect what is on the display?
- How might these inputs affect other components in the system?

In declarative languages like Clock, it is possible to treat these questions separately – a *view* function states what will be on the display; an *invariant* function handles consistency, and event functions handle inputs. It is left up to the compiler to work out when and in what order these functions need to be invoked. In languages with a non-declarative programming model, programmers must work out how every state change might trigger changes in other parts of the system or in the display, and program the changes by hand. The complexity of this task can push programmers into ad-hoc solutions such as simplifying consistency maintenance by making all data global, or simplifying display maintenance by updating the entire display after every modification.

It is critical that the language be *purely* declarative. If constructs such as view or invariant functions permit side-effects, then the programmer must be aware of when they are invoked. If the programmer needs to know what will trigger view and invariant updates, then the benefits of being able to separately deal with inputs, consistency maintenance and view maintenance are lost. As described in chapters 3 and 4, Clock provides declarativeness in the small through the referential transparency property, and declarativeness in the large, guaranteeing that evaluation strategy for view and invariant functions is not visible to Clock programmers.

8.2 Clock and TCFP

The reason why Clock is able to support flexible development of user interfaces within a purely declarative framework is Clock's basis in TCFP. TCFP relaxes the traditional functional programming framework by permitting general I/O and non-determinism. Since TCFP's I/O is completely general, any restricted I/O system (such as that provided in Clock) can be modeled in TCFP.

The design of Clock and TCFP took place in parallel, with much interplay in the design process. The desire to make Clock as flexible as possible led to the improvement of early, more restrictive versions of TCFP. TCFP's formal basis led to confidence that Clock provided the desired declarative properties, and that proposed implementation techniques for Clock would implement the desired semantics.

Therefore, designing the language and its formalism together provided a synergy. If TCFP had not been available to help in exploring language design, it would have been difficult to design a language whose goals are as ambitious as Clock's. Without TCFP, we would not have the same confidence that Clock actually possesses the declarative properties we aimed to provide.

The main lesson to be learned from this design of Clock and TCFP is that language design and language definition are related tasks. While languages should have clean and simple semantics, we should be willing to recognize when our formalisms are too restrictive, and we should be careful not to sacrifice the human factors of languages in order to achieve desirable formal properties.

8.3 The TCFP Framework

TCFP was used to define the semantics of Clock (chapter 6), and provided the framework for reasoning about Clock's properties (chapter 7). While successful for these tasks, one drawback of using TCFP is that formally reasoning about language properties can be arduous. The number of constraints required to describe a language such as Clock is large. The use of abstractions such as streams, wires and flags means that the constraints themselves rely on lower-level constraints. This means that proving even simple properties can rely on extensive manipulation of many logical formulae. Fully formal proofs of such properties as those presented in chapter 7 are in practice difficult. It is clear that some form of mechanical aid would greatly help in using TCFP.

TCFP is based on a very rich temporal logic, embedding all of classical, predicate logic. As yet, practical theorem provers do not exist for logics of this power. One area of further research might be to examine how TCFP can be restricted so that existing theorem provers can be used.

Another difficulty with TCFP is that semantic specifications themselves may be incorrect or incomplete. This is of course a problem with all specification methods. If TCFP were supported by a mechanical theorem prover, it would be possible to easily determine if basic properties of the language hold, thereby increasing confidence in the correctness of the specification. The layered approach of TCFP specifications would aid greatly in this process, since it would be possible to verify properties at one level before proceeding to the next.

Despite these difficulties, TCFP has proven itself to be a practical formalism for expressing the semantics of languages. It is an achievement that it was possible at all to express Clock's semantics. The TCFP method led to extensive interplay between language design, implementation and specification, leading to an improved language design, and increased confidence in the correctness of the implementation.

8.4 Future Work

Clock is already an interesting language for the prototyping of direct manipulation, graphical user interfaces. We are currently investigating a number of directions aiming to improve Clock system itself, and to extend the language to support the development of different styles of user interfaces. These lines of work are:

The Clock Implementation: Clock is currently a compiled language, combining syntactic transforms written in TXL [19] with compilation based on Chakravarty and Lock's GTML compiler [15]. The speed of applications generated with the Clock compiler is fast enough for production use; however, the compilation process itself is slow. An interpretive version of Clock is currently being implemented to allow improved turnaround following modifications in Clock programs. Future plans include the integration of the interpreter with the ClockWorks programming environment to provide a truly flexible user interface development environment.

Fluid Interface Building: One of the drawbacks of programming in Clock is that views must be specified in a purely textual view language. An experimental user interface builder for Clock has been developed [101], allowing views to be drawn rather than described textually. One of the novel aspects of interface building in Clock is that since display views are first class values in Clock, it is easy to mix text and pictures in a single expression. We aim to develop a full editor supporting this style of interface building, where text and graphics can be fluidly mixed.

Multi-User Applications: Multi-user applications support groups of people performing a common task. Examples of multi-user interfaces include multi-user editors, telepresence systems, and collaborative software engineering tools. Clock is currently being extended to support the declarative development of multi-user applications [77].

The key to this development is that Clock architectures provide a high-level model for structuring applications. Components communicate via updates and requests. By implementing these updates and requests over a network, high-level communication between distributed parts of an application can be achieved. The key to this implementation strategy is that Clock's semantics, as described in chapter 6, continue to be hold even when applications have multiple users.

Supporting Multi-Media: Modern user interfaces involve not only text and graphics, but also sound and video. Sound and video (collectively referred to as *continuous media*), differ from traditional text and graphics in that they have temporal aspects – a video or sound clip runs for some amount of time. Programming languages must therefore support the starting, stopping and synchronization of continuous media, as opposed to the atomic operations required to support text and graphics. An interesting challenge is to see whether Clock can be extended to handle temporal media in a high-level, declarative fashion.

Easy Component Reuse: One of the fundamental ideas behind Clock is that user interfaces can be structured via a set of connected components. In order to allow rapid prototyping of interactive systems, it is assumed that a rich library of components is available: ideally, most of the components used in an architecture would be drawn from the library. Even with the small number of components currently available in the Clock library, component reuse is already a major part of development with Clock.

The trouble with component libraries is that as they become large, they rapidly become unmanageable [60]. High-level support is therefore required to help programmers understand what components are available, and which is the best component to use in a given situation. In the future, some such support will have to be developed to help manage component-based programming in Clock.

Clock Methodology: To be effective, software tools must support some software development methodology. The *Clock Methodology* aims to provide a structured framework for user-centred design of interactive systems. Work has already been performed showing how Clock architectures can be derived from task-oriented specifications in the *User Action Notation* [46]. Two significant case studies have been performed with this methodology [23, 67]. Future work involves experimenting with using the Clock methodology throughout the software development process, and investigating tool support to link the phases of the methodology.

8.5 Conclusion

This thesis has demonstrated that it is possible to provide flexible support for developing graphical user interfaces within a purely declarative language based on functional programming. The key to providing a flexible I/O system within a functional language is the use of the novel TCFP framework for specifying and reasoning about extended functional languages. Using TCFP, it was possible to formally define Clock, and to prove Clock's declarative properties.

Appendix A

The Interaction Lambda Calculus

This appendix formally defines the interaction λ-calculus ($I\lambda c$). A less formal introduction is contained in chapter 5.

The interaction λ-calculus ($I\lambda c$) is a small functional programming language extended with constructs for I/O and non-determinism. The basic $I\lambda c$ is a slightly extended version of the traditional lambda calculus. It is shown how through syntactic sugar, the constructs of modern functional programming languages such as Haskell [56] can be easily defined in terms of the constructs of basic $I\lambda c$. This core language is shown to possess the referential transparency property, the key defining property of functional languages.

The basic $I\lambda c$ is extended with constructs to express I/O and non-determinism. $I\lambda c$ has no explicit control strategy, so I/O events are not sequenced. For example, the $I\lambda c$ program

```
(write "stdout" "hello", write "stdout" "world")
```

could produce the output *"hello world"* or *"world hello"*. This means that $I\lambda c$ is not confluent; i.e., for any given $I\lambda c$ redex, there is more than one possible normal form. Since there is no explicit control flow, we can still consider the language to be *declarative*: of the normal forms to which an expression can reduce, all are considered correct.

This chapter introduces the syntax and semantics of the interaction lambda calculus, and concludes with some properties of the language.

A.1 Syntax

The basic core of the Interaction Lambda Calculus (or $I\lambda c$) is the traditional λ-calculus, extended to allow naming of lambda-abstractions (through the `let` construct), and pattern matching (through the `case` construct). This syntax is presented in figure A.1.1.

An $I\lambda c$ program to compute the fibonacci number of 10 is:

```
let
    fib = fn x ->
              case x of
                  0 -> 1
                | 1 -> 1
                | n -> + (fib (- n 2)) (fib (- n 1))
              end case
          end fn
in
    fib 10
end let
```

A.1.1 Syntactic Sugar

The constructs of modern functional languages such as Haskell [56] can be easily added to $I\lambda c$ through syntactic sugar. This approach is used to extend $I\lambda c$ to the full syntax used in chapter 5. For example, an *if* construct:

`if` e_1 `then` e_2 `else` e_3 `end if`

can be defined as:

`case` e_1 `of True ->`e_2 | `False ->`e_3 `end case`

A *let* construct:

`let` $p = e_1$ `in` e_2 `end let`

can be defined as:

`case` e_1 `of` p `->`e_2 `end case`

Equational definitions of functions:

$f\ p_1$ `=` e_1.
...
$f\ p_n$ `=` e_n.
`eval` e.

expn	::=	*variable*
	|	*literal*
	|	*constructorSymbol* { *expn* }
	|	`fn` *variable* `->` *expn* `end fn`
	|	*expn expn*
	|	"(" *expn* ")"
	|	`let` *decl* `in` *expn* `end let`
	|	`case` *expn* `of` *caseBody* `end case`
decl	::=	*variable* = *expn*
	|	*decl* "," *decl*
caseBody	::=	*pat* `->` *expn* {"|" *pat* `->` *expn*}
pat	::=	*variable*
	|	*constructorSymbol* { *pat* }
	|	"(" *pat* ")"

Figure A.1: Abstract Syntax of Basic Interaction Lambda Calculus

can be defined as:

```
let
  f =
    fn x ->
      case x of
        p1 ->e1 |
        ... |
        pn ->en
      end case
    end fn
in
  e
end let
```

In addition, we allow ourselves the use of infix notation for the usual predefined functions such as arithmetic and list construction. The fibonacci example then looks like:

```
fib 0 = 1.
fib 1 = 1.
fib n = fib (n-2) + fib (n-1).
eval fib 10.
```

$$
\begin{array}{rcl}
\mathit{expn} & ::= & \texttt{read}\ \mathit{expn} \\
& | & \texttt{write}\ \mathit{expn}\ \mathit{expn} \\
& | & \texttt{group}\ \mathit{expn}\ \mathit{expn} \\
& | & \texttt{bind}\ \mathit{pat} = \mathit{expn}\ \texttt{in}\ \mathit{expn}\ \texttt{end bind}
\end{array}
$$

Figure A.2: Extensions to Interaction Lambda Calculus

A.1.2 Supporting I/O in $I\lambda c$

In order to give the building blocks for concurrency and interaction, constructs are added to the language for input/output, forcing I/O operation, and grouping I/O events (figure A.1.1). As seen in chapter 5, the *read* construct reads a value from the named communication port, while *write* writes a given value to the named communication port. The *bind* construct is used to force I/O to be performed in an expression, so that the expression has a unique value. An example of the use of *bind* is presented in section 5.2.1. The *group* construct is used to group sets of related I/O operations together. Grouping is used in section 6.3 to sequence threads of I/O activity.

A.2 Semantics

The semantic definition of $I\lambda c$ is very close to that of traditional functional languages. The crucial difference is that the semantic functions are defined in terms of an *ioTrace* parameter that encodes all external behaviour resulting from executing the program. This parameter in effect acts as an oracle: if an input is to take place, the parameter indicates what value is read, over which communication port, and at what time. Outputs are similarly recorded, as are the values of random numbers and explicit sequencing directives. This means that the semantic functions themselves are pure, all the external information is encoded as a parameter.

A.2.1 Semantic Domains

The semantic functions are defined over domains of values, which are (in the traditional style of denotational semantics [96]) structured as complete partial orders.

The values which expressions can take on are built from the basic values: the rational numbers, characters, and boolean values.

$$bv \stackrel{\text{def}}{=} \mathcal{Q}_\perp + char_\perp + \{true, false\}_\perp$$

Values may be constructor terms, represented as a tuple including an identifier for the constructor symbol, and a list of the values being grouped by the constructor:

$$csym \stackrel{\text{def}}{=} I\!N_\perp$$

$$cterm \stackrel{\text{def}}{=} csym \times value^*$$

Values are themselves defined in terms of the basic values, constructor terms, and *interactive functions*, to be described later:

$$value \stackrel{\text{def}}{=} bv + cterm + ifunction$$

An I/O trace is a tree of I/O events that occur over the program's execution. A trace encodes the input, output and random number generation that a program causes, and when these events occur. The tree structure for traces is convenient, as it allows simple splitting of traces to different parts of the functional program.

Events are (analogously to definition B.2.2) tuples encoding a time the event occurred, what the event was (i for input, o for output), on what communication port it took place, and the value and tag communicated:

$$
\begin{array}{rcl}
time & \stackrel{\text{def}}{=} & I\!N \\
commPort & \stackrel{\text{def}}{=} & value \\
uid & \stackrel{\text{def}}{=} & value
\end{array}
$$

$$
\begin{array}{rcl}
event & \stackrel{\text{def}}{=} & time \times \{\mathrm{i}, \mathrm{o}\} \times commPort \times value \times uid \\
events & \stackrel{\text{def}}{=} & \mathcal{P}(event)
\end{array}
$$

I/O traces are a tree of such events, values for random number generation, and I/O sequencing directives:

$$ioTrace \stackrel{\text{def}}{=} event_\perp + ioTrace^*$$

The concept of an *interactive value* can now be described. Interactive values may take on many different values, dependent on the value of a given I/O trace. An interactive value is defined as being a mapping from an I/O trace to a *weak head normal form* (or *whnf*) value. A *whnf* value may be a value, or a constructor term whose components are in term interactive values:

$$
\begin{array}{rcl}
whnf & \stackrel{\text{def}}{=} & value\ +\ csym \times ivalue^* \\
ivalue & \stackrel{\text{def}}{=} & ioTrace \rightarrow whnf
\end{array}
$$

Interactive functions are then considered to be mappings from interactive values onto interactive values:

$$ifunction \stackrel{\text{def}}{=} ivalue \rightarrow ivalue$$

A.2.2 Semantic Functions

We now have the necessary apparatus to define the semantic functions themselves. There are four semantic functions:

$$\begin{array}{lcl} \mathcal{L}[\![\cdot]\!] & : & \mathit{literal} \to \mathit{value} \\ \mathcal{C}[\![\cdot]\!] & : & \mathit{constructorSymbol} \to \mathit{csym} \\ \mathcal{E}[\![\cdot]\!] & : & \mathit{expn} \to \mathit{env} \to \mathit{ivalue} \\ \mathcal{D}[\![\cdot]\!] & : & \mathit{decl} \to \mathit{env} \to \mathit{env} \end{array}$$

The first function assigns meanings to literal expressions, and is assumed to be predefined. The second assigns unique integer identifiers to each of a set of predefined constructor symbols. The third gives meaning to expressions, and the fourth meaning to declarations.

The semantic functions use *environments* to collect definitions, where:

$$\mathit{env} \;:\; \mathit{ident} \to \mathit{ivalue}$$

The *initial environment* μ_0 contains definitions for all predefined constructors and functions, and the undefined value "$\perp$" otherwise.

Letting $a \in \mathit{ident}$, e,e_1,e_2 be *expn* productions, p be a *pat* production, μ be an environment, ι,ι_1, and ι_2 be I/O traces, $d \in \mathit{decl}$, and $b \in \mathit{caseBody}$, then figure A.3 shows the semantics of the basic Interaction Lambda Calculus. Implicitly, the value of any expression not covered by one of these cases is $\perp$. Additionally, the following notes apply to this definition.

In equation A.4, the "_" symbol is a place-holder for the I/O trace parameter. This is meant to emphasize that this parameter is not used in the definition of `fn` abstractions. (I.e., I/O is performed when the function is applied, not when it is defined.)

In equation A.8, the definition function is used to give the values of recursive function definitions. This function is defined as:

$$\begin{array}{l} \mathcal{D}[\![f_1 = e_1, \cdots, f_n = e_n]\!]\,\mu \\ \quad \stackrel{\text{def}}{=}\; \mu\,[f_1 \mapsto \overline{f}_1, \ldots, f_n \mapsto \overline{f}_n] \\ \qquad \text{where } \langle \overline{f}_1, \ldots, \overline{f}_n \rangle = \mathit{fix}\, \lambda\langle \overline{f}_1, \ldots, \overline{f}_n \rangle.\langle \mathcal{E}[\![e_1]\!]\,\mu', \ldots, \mathcal{E}[\![e_n]\!]\,\mu' \rangle \\ \qquad\quad \text{where } \mu' = \mu\,[f_1 \mapsto \overline{f}_1, \ldots, f_n \mapsto \overline{f}_n] \end{array}$$

The semantics of the extensions to Interaction Lambda Calculus are shown in figure A.6, where e, e_1 and e_2 are *expn* productions, μ is an environment, ι_1 and ι_2 are I/O traces, and $n \in I\!N_\perp$. As before, wherever the $\mathcal{E}[\![\cdot]\!]$ function is undefined, it implicitly takes on the "undefined" value "$\perp$".

In these definitions, the I/O trace parameter is used as an external oracle to determine what values are read and written, and the values of random numbers. This behaviour truly has an oracular behaviour, since the input parameter is expected to be in exactly the right format, offering an input tuple every time input is required, an output tuple

$$\begin{array}{rcll} \mathcal{E}[\![l]\!]\,\mu & \stackrel{\text{def}}{=} & \lambda\perp.\mathcal{L}[\![l]\!] & \text{(A.1)} \\ \mathcal{E}[\![v]\!]\,\mu & \stackrel{\text{def}}{=} & \mu\,x & \text{(A.2)} \\ \mathcal{E}[\![c\,e_1\;\ldots\;e_n]\!]\,\mu & \stackrel{\text{def}}{=} & \lambda\perp.\langle \mathcal{C}[\![c]\!], \mathcal{E}[\![e_1]\!]\,\mu, \ldots, \mathcal{E}[\![e_n]\!]\,\mu \rangle & \text{(A.3)} \\ \mathcal{E}[\![\texttt{fn}\;x\;\texttt{->}\;e\;\texttt{end fn}]\!]\,\mu & \stackrel{\text{def}}{=} & \lambda\perp.\lambda v.\mathcal{E}[\![e]\!]\,\mu[x \mapsto v] & \text{(A.4)} \\ \mathcal{E}[\![e_1\,e_2]\!]\,\mu & \stackrel{\text{def}}{=} & \lambda\langle \iota_1, \iota_2 \rangle.(\mathcal{E}[\![e_1]\!]\,\mu\,\iota_1)\,(\mathcal{E}[\![e_2]\!]\,\mu\,)\,\iota_2 & \text{(A.5)} \\ \mathcal{E}[\![(e)]\!] & \stackrel{\text{def}}{=} & \mathcal{E}[\![e]\!] & \text{(A.6)} \\ & & & \text{(A.7)} \\ \mathcal{E}[\![\texttt{let}\;d\;\texttt{in}\;e\;\texttt{end let}]\!]\,\mu & \stackrel{\text{def}}{=} & \mathcal{E}[\![e]\!]\,(\mathcal{D}[\![d]\!]\,\mu\,) & \text{(A.8)} \\ & & & \text{(A.9)} \end{array}$$

$$\begin{array}{rcl} \mathcal{E}[\![\texttt{case}\;e\;\texttt{of} \\ \quad p_1\texttt{->}e_1 \\ \quad \mid\;\ldots \\ \quad \mid\;p_n\texttt{->}e_n \\ \quad \texttt{end case}]\!]\,\mu & \stackrel{\text{def}}{=} & \lambda\langle \iota_1, \ldots, \iota_n \rangle. \\ & & \quad \mathit{let}\;v = \mathcal{E}[\![e]\!]\,\mu\,, \\ & & \qquad \mu_1 = \mathit{match}[\![p_1]\!]\,v\,\iota_1\,\mu, \\ & & \qquad \ldots, \\ & & \qquad \mu_n = \mathit{match}[\![p_n]\!]\,v\,\iota_1\,\mu \\ & & \quad \mathit{in}\;\;(\mathit{if}\;\mu_1 \neq \perp\;\mathit{then} \\ & & \qquad \mathcal{E}[\![e_1]\!]\,\mu_1\,\iota_2 \\ & & \qquad \mathit{elsif}\;\ldots \\ & & \qquad \mathit{elsif}\;\mu_n \neq \perp\;\mathit{then} \\ & & \qquad\quad \mathcal{E}[\![e_n]\!]\,\mu_n\,\iota_2) \end{array} \quad \text{(A.10)}$$

Figure A.3: Semantics of Basic Interaction Lambda Calculus

$$\mathit{match}[\![\cdot]\!] \;:\; \mathit{pat} \to \mathit{ivalue} \to \mathit{ioTrace} \to \mathit{env} \to \mathit{env}$$

$$\begin{array}{rcll} \mathit{match}[\![x]\!]\,v\,\perp\,\mu & \stackrel{\text{def}}{=} & \mu[x \mapsto v] & \text{(A.11)} \\ \mathit{match}[\![(p)]\!] & \stackrel{\text{def}}{=} & \mathit{match}[\![p]\!] & \text{(A.12)} \\ \mathit{match}[\![c\;p_1\;\ldots\;p_n]\!]\,v\,\iota\,\mu & \stackrel{\text{def}}{=} & (\mathit{match}[\![p_1]\!]\,v_1\,\iota_1) \cdots ((\mathit{match}[\![p_n]\!]\,v_n\,\iota_n)\,\mu\,) & \text{(A.13)} \\ & & \quad \text{if } \iota = \langle \iota', \iota_1, \ldots, \iota_n \rangle, \\ & & \qquad \text{and } \langle \mathcal{C}[\![c]\!], v_1, \ldots, v_n \rangle = v\,\iota' \end{array}$$

Figure A.4: Semantics of Pattern Matching

$$eval \;:\; ivalue \to ioTrace \to value$$

$$eval\, v\, \iota \stackrel{\text{def}}{=} v\, \iota \tag{A.14}$$

$$eval\, \langle c, \omega_1, \ldots, \omega_n \rangle\, \langle \iota_1, \ldots, \iota_n \rangle \stackrel{\text{def}}{=} \langle c, eval\, \omega_1\, \iota_1, \ldots, eval\, \omega_n\, \iota_n \rangle \tag{A.15}$$

$$toWhnf \;:\; value \to whnf$$

$$toWhnf\, \langle c, v_1, \ldots, v_n \rangle \stackrel{\text{def}}{=} \langle c, \lambda\iota.toWhnf\ v_1, \ldots, \lambda\iota.toWhnf\ v_n \rangle \tag{A.16}$$

$$toWhnf\ v \stackrel{\text{def}}{=} v \tag{A.17}$$

$$\mathcal{V}[\![\,\cdot\,]\!] \;:\; expn \to env \to ioTrace \to value$$

$$\mathcal{V}[\![e]\!]\, \mu\, \langle \iota_1, \iota_2 \rangle \stackrel{\text{def}}{=} eval\, (\mathcal{E}[\![e]\!]\, \mu\ \iota_1)\, \iota_2 \tag{A.18}$$

Figure A.5: Evaluating Interactive Values

$$\mathcal{E}[\![\texttt{read}\ e]\!]\, \mu \stackrel{\text{def}}{=} \lambda\langle \iota_1, \iota_2 \rangle.toWhnf\ v \tag{A.19}$$
where $\iota_1 = \langle t, \mathrm{i}, c, v, g \rangle$ for some $t : time, v, g : value$,
and $c = \mathcal{V}[\![e]\!]\, \mu\ \iota_2$

$$\mathcal{E}[\![\texttt{write}\ e_1\, e_2]\!]\, \mu \stackrel{\text{def}}{=} \lambda\langle \iota_1, \iota_2, \iota_3 \rangle.toWhnf\ v \tag{A.20}$$
where $\iota_1 = \langle t, \mathrm{o}, c, v, g \rangle$ for some $t : time, g : value$,
$c = \mathcal{V}[\![e_1]\!]\, \mu\ \iota_2$
and $v = \mathcal{V}[\![e_2]\!]\, \mu\ \iota_3$

Figure A.6: Semantics of Extended Interaction Lambda Calculus

for output, and so forth. We can therefore intuitively consider an appropriate I/O trace to be a *model* for an interactive program; in chapter C, exactly this view will be formalized in providing an integrated semantics for $I\lambda c$ with temporal constraints.

In equation A.19, the I/O trace is divided into two parts (ι_1 and ι_2). The first is used to evaluate the expression (e) that is the communication port over which the read is to occur. The second records the time, communication port, value and tag of the input, where the communication port must agree with e.

The definition of *write* in equation A.20 is similar, where the value e_2 is to be written on communication port e_1. The value and communication port recorded in the I/O trace must agree with e_1 and e_2.

A.3 Properties of $I\lambda c$

Traditionally, the defining property of functional languages is referential transparency. Informally, this means that any expression in a functional program always has the same value, regardless of the control strategy being used in executing the program. This notion underlies most reasoning about functional programs: powerful statements can be made about expressions without having to consider the global context in which the expression occurs.

For $I\lambda c$, this definition is naturally extended with the condition that to be referentially transparent, an expression must have the same value regardless of the I/O trace given to the expression. For example, the expression "$2 + 2$" is always equal to 4, whereas the expression

$$2 + (\texttt{read "stdin"})$$

depends on a random number, and therefore will have different values under different executions of the program. "$2 + 2$" is therefore considered to be referentially transparent, whereas "$2 + (\texttt{read "stdin"})$" is not.

We define referential transparency of an expression as follows:

Definition A.3.1 (Referential Transparency)

An expression e is referentially transparent iff there exists $v \in value$ such that for all environments μ and all I/O traces ι,

$$\mathcal{E}[\![e]\!]\, \mu\, \iota \neq \bot \;\Rightarrow\; \mathcal{E}[\![e]\!]\, \mu\, \iota = v$$

□

As we have seen, it is not the case that all $I\lambda c$ expressions are referentially transparent:

Theorem A.3.1 (Referential Transparency of $I\lambda c$)

There exists an $I\lambda c$ expression e that is not referentially transparent.

Proof: By example: the expression "`"stdin"`" does not fulfil definition A.3.1.
□

Fortunately, however, it can be shown that every expression in the basic Interaction Lambda Calculus is referentially transparent.

Theorem A.3.2 (Referential Transparency of Basic $I\lambda c$)

Every expression of Basic $I\lambda c$, as defined in figure A.1.1, is referentially transparent.

Proof: Structural induction over the Basic $I\lambda c$, with reference to the semantics of figure A.3.
□

Two $I\lambda c$ expressions are considered to be equivalent iff they have the same semantics:

Definition A.3.2 (Equivalence of Expressions)

Expressions e_1 and e_2 are *equivalent* iff for all environments μ and all I/O traces ι,

$$\mathcal{E}[\![e_1]\!]\,\mu\,\iota = \mathcal{E}[\![e_2]\!]\,\mu\,\iota$$

□

Many of the syntactic properties of traditional λ-calculus continue to hold. For example, β-reduction is the property that function application is the same as syntactic substitution:

Theorem A.3.3 (Beta Reduction)

For expressions e_1 and e_2, and variable x free in e_1, it holds that:

$$\texttt{fn}\ x\texttt{->}e_1\texttt{end fn}\ e_2 \equiv e_1[e_2/x]$$

Proof: Structural induction over $I\lambda c$.
□

Appendix B

Interaction Logic

As discussed in chapter 5, the interaction λ-calculus is used to *generate* a set of possible program executions, while constraints in the temporal *interaction logic* are used to restrict how programs can execute. Chapter 5 gives examples of how interaction logic can be used to model process connection, synchronization, and persistent data. Chapter 7 shows how interaction logic can be used to describe the properties of languages defined using TCFP. This appendix provides the formal definition of interaction logic.

B.1 Syntax

The atomic formulae of interaction logic are built from symbols, variables, function symbols, and predicate symbols. Function and predicate symbols are assigned a constant rank, specifying the number of arguments they require. The particular set of symbols, variables, and function and predicate symbols from which formulae are constructed is called the *signature* of the logic, defined as follows:

Definition B.1.1 (Signature)

A *signature* of interaction logic is a structure

$$\langle\ \mathit{symbol}, \mathit{variable}, \mathit{expn}, \mathit{predicateSymbol}, r_p\ \rangle$$

where *symbol*, *variable*, and *predicateSymbol* are disjoint alphabets, *expn* is a set of expressions, and $r_p : \mathit{predicateSymbol} \to I\!N$ is a ranking function for the predicate symbols.
□

Referring to the elements of a signature directly can become quite clumsy, so we introduce a series of syntactic short forms:

Convention B.1.1 (Syntactic Short Forms)

Let

$$\Sigma = \langle symbol, variable, functionSymbol, r_f, predicateSymbol, r_p \rangle$$

be a signature. We combine all variables and all symbols by writing:

"x is a Σ–symbol"	for	"$x \in symbol$",
"x is a Σ–variable"	for	"$x \in variable$",
"x is a Σ–atom"	for	"x is a Σ-symbol or x is a Σ–variable", and
"x is a Σ–expression"	for	"$x \in expn$."

We rank predicate symbols by writing:

"p/m is a Σ–predicate symbol" for "$p \in predicateSymbol$ and $r_p(p) = m$."

We leave the signature name Σ implicit when it is clear from context which signature is intended.

□

Terms are either atoms or expressions. As seen in appendix C, expressions are defined by the *Iλc expn* production. Thus, predicates in interaction logic can hold over expressions in some programming language.

Definition B.1.2 (Σ–term)

Let Σ be a signature. Then t is a Σ–*term* iff t is a Σ–atom or t is a Σ–expression.

□

Atomic formulae are built by applying predicate symbols to the required number of terms:

Definition B.1.3 (Σ–Atomic Formula)

Let Σ be a signature, $t_1, \ldots, t_m$ be Σ–terms, and p/m a Σ–predicate symbol. Then $p(t_1, \ldots, t_m)$ is a Σ–atomic formula.

□

Formulae are constructed from atomic formulae and logical connectives. At this point, we introduce the predicates that are predefined in the logic.

Definition B.1.4 (Σ–Formula)

Let

$$\Sigma = \langle symbol, variable, functionSymbol, r_f, predicateSymbol, r_p \rangle$$

be a signature, where the predicate symbols

$$in/3, \quad out/3, \quad =/2$$

are predefined.

Let $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ be Σ–formulae, X a variable, and let X be free in $\mathcal{C}$. Then the following are Σ–formulae:

- All Σ–atomic formulae.
- Constants:

  $TRUE \quad FALSE$
- Classical Connectives:

  $(\mathcal{A}) \qquad \neg\mathcal{A} \qquad \mathcal{A} \wedge \mathcal{B} \qquad \mathcal{A} \vee \mathcal{B}$
  $\mathcal{A} \supset \mathcal{B} \qquad \mathcal{A} \equiv \mathcal{B} \qquad \forall X.\mathcal{C} \qquad \exists X.\mathcal{C}$
- Future Temporal Operators:

  $\bigcirc\mathcal{A} \qquad \Diamond\mathcal{A} \qquad \Box\mathcal{A}$
- Past Temporal Operators:

  $\bullet\mathcal{A} \qquad \blacklozenge\mathcal{A} \qquad \blacksquare\mathcal{A}$
- Future Temporal Relations:

  $\mathcal{A} \trianglelefteq \mathcal{B} \qquad \mathcal{A} \trianglerighteq \mathcal{B} \qquad \mathcal{A} \triangleleft \mathcal{B} \qquad \mathcal{A} \triangleright \mathcal{B} \qquad \mathcal{A} \sqsubseteq \mathcal{B}$
- Past Temporal Relations:

  $\mathcal{A} \blacktriangleleft \mathcal{B} \qquad \mathcal{A} \blacktriangleright \mathcal{B} \qquad \mathcal{A} \blacktriangleleft \mathcal{B} \qquad \mathcal{A} \blacktriangleright \mathcal{B} \qquad \mathcal{A} \sqsubseteq \mathcal{B}$

□

B.1.1 Precedence

Formulae can always be disambiguated using parentheses. Since extensive use of parentheses is syntactically cumbersome, we introduce the following rules of precedence among connectives:

Definition B.1.5 (Precedence)

The connectives bind in the following order, from most to least tightly binding. Connectives appearing at the same level bind from left to right.

$$\neg,\ \bigcirc,\ \Diamond,\ \Box,\ \bullet,\ \blacklozenge,\ \blacksquare$$
$$\trianglelefteq,\ \trianglerighteq,\ \lhd,\ \rhd,\ \sqsubseteq,\ \blacktriangleleft,\ \blacktriangleright,\ \blacktriangleleft,\ \blacktriangleright,\ \blacksquare$$
$$\wedge$$
$$\vee$$
$$\supset$$
$$\equiv$$
$$\forall,\ \exists$$

□

B.1.2 Introducing Sorts

Interaction logic is fundamentally one-sorted: functions and predicates are defined over all atoms and terms. Our examples have shown, however, that functions and predicates generally have some intended domain that is smaller than the entire universe of possible values. For example, in the formula

$$in(c, v, i)$$

it is intuitively intended that c be a communication port, v a value, and i a tag. When using *in*, this intention can be expressed using predicates $commPort/1$, $value/1$, and $tag/1$:

$$in(c, v, i) \wedge commPort(c) \wedge value(v) \wedge tag(i)$$

This approach is somewhat clumsy, so we introduce syntactic sugar to allow the definition of predicates and quantification over restricted domains. These restricted domains we shall call sorts.

Quantification over sorts is defined as:

Definition B.1.6 (Sorted Quantification)

If Σ is a signature, x is a Σ–variable, $s/1$ is a Σ–predicate symbol, and $\mathcal{A}$ is a Σ–formula where s is free in $\mathcal{A}$, then

$$\forall x : s.\mathcal{A}$$

is syntactic sugar for

$$\forall x.(s(X) \supset \mathcal{A}))$$

Similarly,

$$\exists x : s.\mathcal{A}$$

is syntactic sugar for

$$\exists x.(s(x) \wedge \mathcal{A}))$$

□

Predicates over sorts are defined as:

Definition B.1.7 (Sorted Predicates)

If Σ is a signature, $p/m, s_1/1, \ldots, s_m/1$ are Σ–predicate symbols, $x_1, \ldots, x_m$ are Σ–variables, and $\mathcal{A}$ is a Σ–formula where $x_1, \ldots, x_m$ are free in $\mathcal{A}$, then

$$p(x_1 : s_1, \ldots, x_m : s_m) \equiv \mathcal{A}$$

is syntactic sugar for

$$\forall x_1 : s_1. \cdots \forall x_m : s_m.(p(x_1, \ldots, x_m) \equiv \mathcal{A})$$

□

Note that according to this definition, predicates are still defined over all values, and do not hold when applied to values of the wrong sort.

Example B.1.1 (Sort Notation)

A less specific version of the *in* predicate can be defined that is true for a particular communication port in a particular time iff any input event at all has occured on that channel in that time:

$$in_c(c : commPort) \equiv \exists v : value.(\exists i : tag.(in(c, v, i))$$

□

B.2 Semantics

Formulae of interaction logic are based over linear, discrete time. Intuitively, this means that there is always exactly one future, and that time moves incrementally from one position to the next. Formally, we model time with the natural numbers – time has a "start" (i.e., time 0), and no definite end. Time is totally ordered, hence linear; the natural numbers are discrete, and thus so is time.

Definition B.2.1 (Time)

We define the set *time* as the natural numbers:

$$time \stackrel{\text{def}}{=} I\!N$$

□

We first define the concept of *interpretation*, a structure forming the basis of the semantics, and then derive the *interpretation function*, giving meaning to formulae.

The semantics of formulae are based on two sets: a universe of *values* and a universe of *events*. The values are the names of communication ports, and the values and tags that can be sent over these ports. Events are a temporal recording of *what* has been sent over which ports and *when*. In addition to these sets, an interpretation includes functions that give meaning to function and predicate symbols and atoms.

Definition B.2.2 (Σ–Interpretation)

Let Σ be a signature. A Σ–*interpretation* is a structure

$$\langle V, E, \alpha, \phi, \rho_m \rangle$$

where:

- V is a universe of values.
- E is a universe of events.
- The function
  $$\alpha \ : \ \Sigma\text{–}atoms \to time \to V$$
  maps an atom to its value at a particular time point.
- The functions
  $$\phi \ : \ \Sigma\text{–}expression \to V$$
  $$\rho_m \ : \ \Sigma\text{–}predicateSymbol \to time \to \mathcal{P}(V^m)$$
  are families of functions giving meaning to function and predicate symbols of the corresponding arity.

Events are tuples of the form:

$$\langle t, k, c, v, i \rangle$$

where

- $k \in \{\mathtt{r}, \mathtt{w}\}$ (for literal symbols `r` and `w`),
- $t \in time$,
- $c, v, i \in V$, represent a communication port, value and tag respectively.

□

Some properties of the preceding definition are worth noting. First, there is only one universe of values. This means that all communication ports, values, and tags must fit in this universe. When we apply this logic to a real programming language (in chapter A), typically all elements of V will be values, and subsets of V will play dual roles as communication ports or tags.

Secondly, note that we have now defined the concept of *event*. Intuitively, an event consists of something happening over one of the communication ports used by the program. An event can either be a *read* or *write* on the port (corresponding to *in* and *out* in the logic respectively). Events are time-stamped, and the port, value, and tag are given.

This gives us the necessary machinery to give meanings to formulae. Note that in the meta-language used to define formulae, we use the traditional connectives of classical logic. To distinguish these from the notation of the object language, we write these as: *and* (conjunction), *or* (disjunction), *not* (negation), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence), *for all* (universal quantification), *exists* (existential quantification), t (truth), f (falsehood). These have the traditional definitions of classical logic, and the traditional theorems hold (see for example [35]).

Definition B.2.3 ($\mathcal{I}$–Interpretation Function)

Let Σ be a signature, and

$$\mathcal{I} = \langle V, E, \alpha, \phi_n, \rho_m, is_v, is_c, is_t \rangle$$

be a Σ–interpretation. We first define a helper function:

$$\mathcal{T}[\![\cdot]\!] \ : \ \Sigma\text{–}term \to time \to V$$

Let a be a Σ–atom, $t_1, \ldots, t_n$ Σ–terms, f/n a Σ–function symbol, and $\tau \in time$. Then:

$$\mathcal{T}[\![a]\!] \stackrel{\text{def}}{=} \alpha(a)$$
$$\mathcal{T}[\![f(t_1, \ldots, t_n)]\!]\tau \stackrel{\text{def}}{=} \phi_n(f)\langle \mathcal{T}[\![t_1]\!]\tau, \ldots, \mathcal{T}[\![t_n]\!]\tau \rangle$$

We define $\mathcal{I}[\![\,\cdot\,]\!]$, the $\mathcal{I}$–interpretation function, where

$$\mathcal{I}[\![\,\cdot\,]\!] \;:\; \Sigma\text{–}formulae \to time \to \{t, f\}$$

as follows. Let p/m be a Σ–predicate symbol, $t_1, \ldots, t_m$ Σ–terms, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ and Σ–formulae, X a variable where X is free in $\mathcal{D}$, $\tau \in time$, and x a variable ranging over V. Then:

- Σ–atomic formulae:

$$\mathcal{I}[\![p(t_1, \ldots, t_m)]\!]\tau \stackrel{\text{def}}{=} \rho_m\,(p)(\tau)\langle \mathcal{T}[\![t_1]\!]\tau, \ldots, \mathcal{T}[\![t_1]\!]\tau\rangle$$

- Constants:

$$\mathcal{I}[\![TRUE]\!]\tau \stackrel{\text{def}}{=} t$$
$$\mathcal{I}[\![FALSE]\!]\tau \stackrel{\text{def}}{=} f$$

- Classical Connectives:

$$\mathcal{I}[\![(\mathcal{A})]\!]\tau \stackrel{\text{def}}{=} \mathcal{I}[\![\mathcal{A}]\!]\tau$$
$$\mathcal{I}[\![\neg\mathcal{A}]\!]\tau \stackrel{\text{def}}{=} not\ \mathcal{I}[\![\mathcal{A}]\!]\tau$$
$$\mathcal{I}[\![\mathcal{A} \wedge \mathcal{B}]\!]\tau \stackrel{\text{def}}{=} \mathcal{I}[\![\mathcal{A}]\!]\tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau$$
$$\mathcal{I}[\![\mathcal{A} \vee \mathcal{B}]\!]\tau \stackrel{\text{def}}{=} \mathcal{I}[\![\mathcal{A}]\!]\tau\ or\ \mathcal{I}[\![\mathcal{B}]\!]\tau$$
$$\mathcal{I}[\![\mathcal{A} \supset \mathcal{B}]\!]\tau \stackrel{\text{def}}{=} \mathcal{I}[\![\mathcal{A}]\!]\tau \Rightarrow \mathcal{I}[\![\mathcal{B}]\!]\tau$$
$$\mathcal{I}[\![\mathcal{A} \equiv \mathcal{B}]\!]\tau \stackrel{\text{def}}{=} \mathcal{I}[\![\mathcal{A}]\!]\tau \Leftrightarrow \mathcal{I}[\![\mathcal{B}]\!]\tau$$
$$\mathcal{I}[\![\forall X.\mathcal{D}]\!]\tau \stackrel{\text{def}}{=} for\ all\ x \in V.(\mathcal{I}'[\![\mathcal{D}]\!]\tau)$$
where $\mathcal{I}'[\![\,\cdot\,]\!]$ is the $\mathcal{I}'$–interpretation function,
where $\mathcal{I}' = \mathcal{I}$ when $\alpha[X \mapsto x]$ is subst'd for α
$$\mathcal{I}[\![\exists X.\mathcal{D}]\!]\tau \stackrel{\text{def}}{=} exists\ x \in V.(\mathcal{I}'[\![\mathcal{D}]\!]\tau)$$
where $\mathcal{I}'[\![\,\cdot\,]\!]$ is the $\mathcal{I}'$–interpretation function,
where $\mathcal{I}' = \mathcal{I}$ when $\alpha[X \mapsto x]$ is subst'd for α

- Future Temporal Operators:

$$\mathcal{I}[\![\,\bigcirc\mathcal{A}]\!]\tau \stackrel{\text{def}}{=} \mathcal{I}[\![\mathcal{A}]\!](\tau + 1)$$
$$\mathcal{I}[\![\,\Diamond\mathcal{A}]\!]\tau \stackrel{\text{def}}{=} \exists\tau'.(\tau' > \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau')$$
$$\mathcal{I}[\![\,\Box\mathcal{A}]\!]\tau \stackrel{\text{def}}{=} \forall\tau'.(\tau' > \tau \Rightarrow \mathcal{I}[\![\mathcal{A}]\!]\tau')$$

- Past Temporal Operators:

$$\mathcal{I}[\![\,\bullet\mathcal{A}]\!]\tau \stackrel{\text{def}}{=} \mathcal{I}[\![\mathcal{A}]\!](\tau - 1)$$
$$\mathcal{I}[\![\,\blacklozenge\mathcal{A}]\!]\tau \stackrel{\text{def}}{=} \exists\tau'.(\tau' < \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau')$$
$$\mathcal{I}[\![\,\blacksquare\mathcal{A}]\!]\tau \stackrel{\text{def}}{=} \forall\tau'.(\tau' < \tau \Rightarrow \mathcal{I}[\![\mathcal{A}]\!]\tau')$$

- Future Temporal Relations:

$$\mathcal{I}[\![\mathcal{A} \trianglelefteq \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_b.(\tau_b \geq \tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b)$$
$$\Rightarrow (exists\ \tau_a.(\tau_a \geq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a$$
$$and\ for\ all\ \tau_b.(\tau \leq \tau_b < \tau_a \Rightarrow not\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b)))$$
$$\mathcal{I}[\![\mathcal{A} \trianglerighteq \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_a.(\tau_a \geq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a)$$
$$\Rightarrow (exists\ \tau_b.(\tau_b \geq \tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b$$
$$and\ for\ all\ \tau_a.(\tau \leq \tau_a < \tau_b \Rightarrow not\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a)))$$

$$\mathcal{I}[\![\mathcal{A} \triangleleft \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_a.(\tau_a \geq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a$$
$$and\ for\ all\ \tau_b.(\tau \leq \tau_b \leq \tau_a \Rightarrow not\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b))$$
$$\mathcal{I}[\![\mathcal{A} \triangleright \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_b.(\tau_b \geq \tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b$$
$$and\ for\ all\ \tau_a.(\tau \leq \tau_a \leq \tau_b \Rightarrow not\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a))$$
$$\mathcal{I}[\![\mathcal{A} \sqsupset\ \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$(not\ exists\ \tau_a.(\tau_a \geq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a)$$
$$and\ not\ exists\ \tau_b.(\tau_b \geq \tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b))$$
$$or\ exists\ \tau'.(\tau' \geq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau'\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau'$$
$$and\ not\ exists\ \tau''.(\tau \leq \tau'' < \tau'\ and\ (\mathcal{I}[\![\mathcal{A}]\!]\tau'\ or\ \mathcal{I}[\![\mathcal{B}]\!]\tau')))$$

- Past Temporal Relations:

$$\mathcal{I}[\![\mathcal{A} \blacktriangleleft \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_b.(\tau_b \leq \tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b)$$
$$\Rightarrow (exists\ \tau_a.(\tau_a \leq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a$$
$$and\ for\ all\ \tau_b.(\tau_a < \tau_b \leq \tau \Rightarrow not\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b)))$$
$$\mathcal{I}[\![\mathcal{A} \blacktriangleright \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_a.(\tau_a \leq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a)$$
$$\Rightarrow (exists\ \tau_b.(\tau_b \leq \tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b$$
$$and\ for\ all\ \tau_a.(\tau_b < \tau_a \leq \tau \Rightarrow not\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a)))$$
$$\mathcal{I}[\![\mathcal{A} \blacktriangleleft \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_a.(\tau_a \leq \tau\ and\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a$$
$$and\ for\ all\ \tau_b.(\tau_a \leq \tau_b \leq \tau \Rightarrow not\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b))$$
$$\mathcal{I}[\![\mathcal{A} \blacktriangleright \mathcal{B}]\!]\tau \stackrel{\text{def}}{=}$$
$$exists\ \tau_b.(\tau_b \leq \tau\ and\ \mathcal{I}[\![\mathcal{B}]\!]\tau_b$$
$$and\ for\ all\ \tau_a.(\tau_b \leq \tau_a \leq \tau \Rightarrow not\ \mathcal{I}[\![\mathcal{A}]\!]\tau_a))$$

$$\begin{array}{l}\mathcal{I}[\![\mathcal{A} \blacksquare \mathcal{B}]\!]\tau \stackrel{\text{def}}{=} \\ \quad (\textit{not exists } \tau_a.(\tau_a \leq \tau \textit{ and } \mathcal{I}[\![\mathcal{A}]\!]\tau_a) \\ \qquad \textit{and not exists } \tau_b.(\tau_b \leq \tau \textit{ and } \mathcal{I}[\![\mathcal{B}]\!]\tau_b)) \\ \quad \textit{or exists } \tau'.(\tau' \leq \tau \textit{ and } \mathcal{I}[\![\mathcal{A}]\!]\tau' \textit{ and } \mathcal{I}[\![\mathcal{B}]\!]\tau' \\ \qquad \textit{and not exists } \tau''.(\tau < \tau'' \leq \tau' \textit{ and } (\mathcal{I}[\![\mathcal{A}]\!]\tau' \textit{ or } \mathcal{I}[\![\mathcal{B}]\!]\tau')))\end{array}$$

- Predefined Predicates:

$$\begin{array}{rcl}\mathcal{I}[\![in(\mathcal{A},\mathcal{B},\mathcal{C})]\!]\tau & \stackrel{\text{def}}{=} & \langle \tau, \mathtt{r}, \mathcal{I}[\![\mathcal{A}]\!]\tau, \mathcal{I}[\![\mathcal{B}]\!]\tau, \mathcal{I}[\![\mathcal{C}]\!]\tau\rangle \in E \\ \mathcal{I}[\![out(\mathcal{A},\mathcal{B},\mathcal{C})]\!]\tau & \stackrel{\text{def}}{=} & \langle \tau, \mathtt{w}, \mathcal{I}[\![\mathcal{A}]\!]\tau, \mathcal{I}[\![\mathcal{B}]\!]\tau, \mathcal{I}[\![\mathcal{C}]\!]\tau\rangle \in E \\ \mathcal{I}[\![t_1 = t_2]\!]\tau & \stackrel{\text{def}}{=} & \mathcal{T}[\![t_1]\!]\tau = \mathcal{T}[\![t_2]\!]\tau\end{array}$$

□

An interpretation is called a *model* for a set of formulae if it satisfies all the formulae over all time:

Definition B.2.4 (Model)

Let Σ be a signature, $\mathcal{F}$ be a set of Σ–formulae, $\mathcal{I}$ a Σ–interpretation, and $\mathcal{I}[\![\,\cdot\,]\!]$ the $\mathcal{I}$–interpretation function. We say $\mathcal{I}$ *is a model for* $\mathcal{F}$, written

$$\mathcal{I} \models \mathcal{F}$$

iff for all $\mathcal{A} \in \mathcal{F}$ and all τ in *time*,

$$\mathcal{I}[\![\mathcal{A}]\!]\tau = t$$

□

A tautology is then a formula which always holds.

Definition B.2.5 (Tautology)

Let Σ be a signature, and $\mathcal{A}$ a Σ–formula. We say $\mathcal{A}$ *is a tautology*, written

$$\vdash \mathcal{A}$$

Iff for every Σ–interpretation $\mathcal{I}$,

$$\mathcal{I} \models \{\mathcal{A}\}$$

□

B.2.1 A Core Logic

When defining a logic, it is traditional to define only a small core of logical connectives, and to define the remaining connectives as syntactic sugar building on the core. For example, in classical logic, disjunction is often defined as:

$$\mathcal{A} \vee \mathcal{B} \stackrel{\text{def}}{=} \neg(\neg\mathcal{A} \wedge \neg\mathcal{B})$$

As well as conciseness, this approach has the advantage that proofs requiring induction over the structure of formulae need only consider the core connectives.

'The disadvantage of defining connectives syntactically is that there is no assurance that the definitions correspond to the intuitive definitions one expects. For example, with the non-standard semantics of interaction logic, can we be sure that the above definition of disjunction corresponds to our intution of what disjunction should mean?

Our approach has been to laboriously define all of the connectives, so that our intuition is satisfied. We now show that the logic could have been built from a small set of connectives (in particular, $\wedge$, $\neg$, $\forall$, $\bigcirc$, $\bullet$, $\lhd$, and $\rhd$), by proving the syntactic equivalences as a theorem. This approach gives us the advantage of being sure that our definitions are intuitively correct (since all connectives are defined directly in the semantics), and also of having a core logic over which we can reason. We shall call this logic the *Core Interaction Logic*.

Theorem B.2.1 (Core Equivalences)

Let Σ be a signature, $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ be Σ–formulae, X a Σ–variable where X is free in $\mathcal{C}$, and $p/0$ a Σ–predicate. Then each of the following formulae is a tautology:

$$\begin{array}{rcl}\mathit{FALSE} & \equiv & p \wedge \neg p \\ \mathit{TRUE} & \equiv & \neg\mathit{FALSE}\end{array}$$

$$\begin{array}{rcl}\mathcal{A} \vee \mathcal{B} & \equiv & \neg(\neg\mathcal{A} \wedge \neg\mathcal{B}) \\ \mathcal{A} \supset \mathcal{B} & \equiv & \neg\mathcal{A} \vee \mathcal{B} \\ (\mathcal{A} \equiv \mathcal{B}) & \equiv & (\mathcal{A} \supset \mathcal{B}) \wedge (\mathcal{B} \supset \mathcal{A})\end{array}$$

$$(\exists X.\mathcal{C}) \equiv (\neg\forall X.\neg\mathcal{C})$$

$$\begin{array}{rcl}\Diamond\mathcal{A} & \equiv & \bigcirc(\mathcal{A} \lhd \mathit{FALSE}) \\ \blacklozenge\mathcal{A} & \equiv & \bullet(\mathcal{A} \blacktriangleleft \mathit{FALSE}) \\ \Box\mathcal{A} & \equiv & \neg\Diamond\neg\mathcal{A} \\ \blacksquare\mathcal{A} & \equiv & \neg\blacklozenge\neg\mathcal{A}\end{array}$$

$$\begin{array}{rcl}\mathcal{A} \sqsubset \mathcal{B} & \equiv & \neg(\mathcal{A} \lhd \mathcal{B} \vee \mathcal{B} \lhd \mathcal{A}) \\ \mathcal{A} \trianglelefteq \mathcal{B} & \equiv & \mathcal{A} \lhd \mathcal{B} \vee \mathcal{A} \sqsubset \mathcal{B}\end{array}$$

$$\mathcal{A} \rhd \mathcal{B} \equiv \mathcal{B} \lhd \mathcal{A}$$
$$\mathcal{A} \trianglerighteq \mathcal{B} \equiv \mathcal{B} \trianglelefteq \mathcal{A}$$

Proof: These equivalences follow directly from definition B.2.3.
□

B.3 Proofs In Interaction Logic

In order to use the logic effectively, we need to reason *within* the logic, that is, without having to resort to its semantics. We introduce a calculus for the logic, based on a set of inference rules and axiom schemata. The inference rules allow us to deduce the truth of formulae from formulae we already know to be true, and the axiom schemata provide us with a stock of tautologies to start with.

We show that the rules of inference are correct, and that the axioms indeed always hold – this leads to a soundness theorem for the calculus (theorem B.3.1.) A second desirable property of a calculus is completeness – i.e., that we have all the rules and axioms we need to be able to prove any true theorem of the logic. Unfortunately, it is often the case with temporal logics that no such complete calculus exists [1], which is the case in interaction logic as well. The practical result of this incompleteness is that no matter how many axioms we add to the calculus, there will still be theorems that cannot be proved without adding at least one axiom. The calculus is, however, quite powerful, and is sufficient to prove all the results contained in this report.

We first define the notion of entailment, and then present the rules of inference and axiom schemata of interaction logic.

Definition B.3.1 (Entailment)

If Σ is a signature, we say that a set of Σ–formulae Γ *entails* the Σ–formula $\mathcal{A}$, written

$$\Gamma \vdash \mathcal{A}$$

iff $\mathcal{A}$ can be obtained from Γ and the axioms of interaction logic through the inference rules of interaction logic.

□

The rules of inference for the logic are shown in figure B.1. The first twelve rules (i.e., those without $\Box$-Introduction and $\blacksquare$-Introduction) form the traditional Gentzen calculus, which when augmented with the axiom $\vdash \mathcal{A} \vee \neg\mathcal{A}$ provides a complete calculus for first order predicate logic. These rules are as presented in [27], but specified in a notation closer to that of [34]. The fact that these rules also form the basis of a calculus

$$\frac{\Gamma \vdash \mathcal{A} \quad \Gamma \vdash \mathcal{B}}{\Gamma \vdash \mathcal{A} \wedge \mathcal{B}} \quad (\wedge\textit{-Introduction})$$

$$\frac{\Gamma \vdash \mathcal{A}}{\Gamma \vdash \mathcal{A} \vee \mathcal{B}} \quad \frac{\Gamma \vdash \mathcal{B}}{\Gamma \vdash \mathcal{A} \vee \mathcal{B}} \quad (\vee\textit{-Introduction})$$

$$\frac{\Gamma \vdash \mathcal{A} \wedge \mathcal{B}}{\Gamma \vdash \mathcal{A}} \quad \frac{\Gamma \vdash \mathcal{A} \wedge \mathcal{B}}{\Gamma \vdash \mathcal{B}} \quad (\wedge\textit{-Elimination})$$

$$\frac{\Gamma \vdash \mathcal{A} \vee \mathcal{B} \quad \Gamma, \mathcal{A} \vdash \mathcal{C} \quad \Gamma, \mathcal{B} \vdash \mathcal{C}}{\Gamma \vdash \mathcal{C}} \quad (\vee\textit{-Elimination})$$

$$\frac{\Gamma, \mathcal{A} \vdash \mathcal{B}}{\Gamma \vdash \mathcal{A} \supset \mathcal{B}} \quad (\supset\textit{-Introduction})$$

$$\frac{\Gamma \vdash \mathcal{A} \quad \Gamma \vdash \mathcal{A} \supset \mathcal{B}}{\Gamma \vdash \mathcal{B}} \quad (\supset\textit{-Elimination})$$

$$\frac{\Gamma \vdash \mathcal{A}}{\Gamma \vdash \forall X.(\mathcal{A}[a \mapsto X])} \quad (\forall\textit{-Introduction})$$

$$\frac{\Gamma \vdash \forall X.\mathcal{A}}{\Gamma \vdash (\mathcal{A}[X \mapsto a])} \quad (\forall\textit{-Elimination})$$

$$\frac{\Gamma \vdash \mathcal{A}}{\Gamma \vdash \exists X.(\mathcal{A}[a \mapsto X])} \quad (\exists\textit{-Introduction})$$

$$\frac{\Gamma \vdash \exists X.\mathcal{A} \quad \Gamma, \mathcal{A}[X \mapsto a] \vdash \mathcal{B}}{\Gamma \vdash \mathcal{B}} \quad (\exists\textit{-Elimination})$$

$$\frac{\Gamma, \mathcal{A} \vdash \textit{FALSE}}{\Gamma \vdash \neg\mathcal{A}} \quad (\neg\textit{-Introduction})$$

$$\frac{\Gamma \vdash \mathcal{A} \quad \Gamma \vdash \neg\mathcal{A}}{\Gamma \vdash \textit{FALSE}} \quad \frac{\Gamma \vdash \textit{FALSE}}{\Gamma \vdash \mathcal{A}} \quad (\neg\textit{-Elimination})$$

$$\frac{\vdash \mathcal{A}}{\vdash \Box\mathcal{A}} \quad (\Box\textit{-Introduction})$$

$$\frac{\vdash \mathcal{A}}{\vdash \blacksquare\mathcal{A}} \quad (\blacksquare\textit{-Introduction})$$

where $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are Σ–formulae for some signature Σ, X is a Σ–variable, a is a Σ–symbol, and in the rules $\forall$–Introduction and Elimination, and $\exists$–Introduction and Elimination, a is free in $\mathcal{A}$, and for all $f \in \Gamma$, a is not free in f.

Figure B.1: Inference Rules of Interaction Logic

for interaction logic is reassuring, since all of the familiar theorems of classical logic are also theorems of interaction logic.

In addition to the rules of inference, we start with a set of formulae which we know to be true. We call these formulae the *initial axioms* of interaction logic.

Definition B.3.2 (Initial Axioms)

For Σ a signature, and $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{B}$ and $\mathcal{B}'$ Σ–formulae, the following axiom schemata are defined to be tautologies:

- All the equivalences of theorem B.2.1.
- Law of excluded middle:

  $$\vdash \mathcal{A} \vee \neg\mathcal{A}$$

- Definition of $\bigcirc$ and $\bullet$:

  $$\vdash \neg \bigcirc \mathcal{A} \equiv \bigcirc \neg \mathcal{A}$$
  $$\vdash \bigcirc(\mathcal{A} \supset \mathcal{B}) \equiv (\bigcirc \mathcal{A} \supset \bigcirc \mathcal{B})$$

  $$\vdash \neg \bullet \mathcal{A} \equiv \bullet \neg \mathcal{A}$$
  $$\vdash \bullet(\mathcal{A} \supset \mathcal{B}) \equiv (\bullet \mathcal{A} \supset \bullet \mathcal{B})$$

- Definition of $\lhd$:

  $$\vdash \mathcal{A} \lhd \mathcal{B} \wedge (\mathcal{A} \supset \mathcal{A}') \supset \mathcal{A}' \lhd \mathcal{B}$$
  $$\vdash \mathcal{A} \lhd \mathcal{B} \wedge (\mathcal{B} \supset \mathcal{B}') \supset \mathcal{A} \lhd \mathcal{B}'$$
  $$\vdash \mathcal{A} \lhd \mathcal{B} \wedge \mathcal{B} \lhd \mathcal{C} \supset \mathcal{A} \lhd \mathcal{C}$$
  $$\vdash \neg(\mathcal{A} \lhd \mathcal{B})$$
  $$\vdash \neg(\mathcal{A} \lhd \mathcal{B}) \equiv \mathcal{A} \trianglerighteq \mathcal{B}$$
  $$\vdash \mathcal{A} \lhd \mathcal{B} \equiv \neg\mathcal{B} \wedge (\mathcal{A} \vee \bigcirc(\mathcal{A} \lhd \mathcal{B}))$$

- Definition of $\blacktriangleleft$:

  $$\vdash \mathcal{A} \blacktriangleleft \mathcal{B} \wedge (\mathcal{A} \supset \mathcal{A}') \supset \mathcal{A}' \blacktriangleleft \mathcal{B}$$
  $$\vdash \mathcal{A} \blacktriangleleft \mathcal{B} \wedge (\mathcal{B} \supset \mathcal{B}') \supset \mathcal{A} \blacktriangleleft \mathcal{B}'$$
  $$\vdash \mathcal{A} \blacktriangleleft \mathcal{B} \wedge \mathcal{B} \blacktriangleleft \mathcal{C} \supset \mathcal{A} \blacktriangleleft \mathcal{C}$$
  $$\vdash \neg(\mathcal{A} \blacktriangleleft \mathcal{B})$$
  $$\vdash \neg(\mathcal{A} \blacktriangleleft \mathcal{B}) \equiv \mathcal{A} \trianglerighteq \mathcal{B}$$
  $$\vdash \mathcal{A} \blacktriangleleft \mathcal{B} \equiv \neg\mathcal{B} \wedge (\mathcal{A} \vee \bullet(\mathcal{A} \blacktriangleleft \mathcal{B}))$$

□

B.3.1 Properties of the Calculus

It is necessary to show that usings the axioms and rules of this calculus, that only true formulae can be deduced, i.e. that no false formulae can be proved to be true. This property is called *soundness*.

Theorem B.3.1 (Soundness)

Let Σ be a signature, $\mathcal{F}$ a set of Σ–formulae, and $\mathcal{A}$ a Σ–formula. Then

$$\mathcal{F} \vdash \mathcal{A}$$

only if for all Σ–interpretations $\mathcal{I}$,

$$\mathcal{I} \models \mathcal{F} \Rightarrow \mathcal{I} \models \mathcal{A}$$

Proof: The proof proceeds by showing that each of the initial axioms from definition B.3.2 are tautologies, and that each of the inference rules of figure B.1 preserves the entailment relation.

□

An additional property that is desirable of a calculus, but not necessary, is that any true formula can be deduced with the calculus. This property is called *completeness*, and is defined as follows:

Definition B.3.3 (Completeness)

Let Σ be a signature, $\mathcal{F}$ a set of Σ–formulae, and $\mathcal{A}$ a Σ–formula. We say the entailment relation $\vdash$ is *complete* iff whenever it holds that for all Σ–interpretations $\mathcal{I}$,

$$\mathcal{I} \models \mathcal{F} \Rightarrow \mathcal{I} \models \mathcal{A}$$

then

$$\mathcal{F} \vdash \mathcal{A}$$

□

As is the case of all interesting temporal logics, interaction logic does not possess a complete proof calculus. This can be proved by constructing a sentence in the logic which completely defines the natural numbers and their associated operations. By the Gödel incompleteness theorem, there is no calculus which can prove all the consequences of this sentence. One presentation of this proof can be found in [1].

Appendix C

Combining Temporal Constraints and Functional Programming

The preceding two chapters have introduced *Interaction Logic*, a temporal logic for expressing constraints over the interactive behaviour of functional programs, and the *Interaction Lambda Calculus*, a functional language extended with constructs to express I/O and non-determinism. The semantics of both of these systems was precisely specified.

This section introduces a combined semantic framework, where the semantics of interaction logic are combined with the $I\lambda c$ semantics. This allows meaning to be given to TCFP programs, involving both functions and temporal constraints. Once the semantics of TCFP programs has been given, we define the concept of semantic-preserving transformations over TCFP programs.

To create the combined semantics, we first cast the semantics of $I\lambda c$ programs into a model theoretic framework. The combined semantics is then the composition of the model theoretic semantics of $I\lambda c$ and that of interaction logic.

C.1 A Model Theoretic Interpretation for $I\lambda c$

In order to combine the semantics of $I\lambda c$ with that of interaction logic, we need to consider $I\lambda c$ as being a logic as well. We can then consider the coombined semantics of an $I\lambda c$ program and a set of constraints as being the intersection of all models for the program with all models for the constraints.

Intuitively, we define a model as being a possible input trace to a program that produces some sensible (i.e., defined) output. This allows us to use the $\mathcal{E}[\![\,\cdot\,]\!]$ function as an interpretation function, where the I/O trace parameter is a model for the program.

We first define the notion of a program signature, simply as being a set of variables, constructor symbols and function symbols to be used in the program:

Definition C.1.1 ($I\lambda c$–Signature)

A signature of the interactive lambda calculus is a structure

$$\langle \mathcal{V}, \mathcal{C}, r_c, \mathcal{F} \rangle$$

where $\mathcal{V}$, $\mathcal{C}$, and $\mathcal{F}$ are disjoint alphabets of symbols, and

$$r_c : \mathcal{C} \to I\!N$$

is a ranking function assigning an arity to constructor symbols.

□

A program is then constructed from a signature using the grammar rules of the preceding chapter:

Definition C.1.2 (Γ–Program)

Given an $I\lambda c$–signature

$$\Gamma = \langle \mathcal{V}, \mathcal{C}, r_c, \mathcal{F} \rangle$$

we define $\mathcal{P}$ to be a Γ–program if $\mathcal{P}$ satisfies the grammar of figures A.1.1 and A.1.1, where the non-terminals *variable*, *constructorSymbol*, and *functionSymbol* take on the values of $\mathcal{V}$, $\mathcal{C}$, and $\mathcal{F}$ respectively.

□

An interpretation for a program then contains a set of events that results from the execution of the program, and the value of the program. Additionally, the meaning of the function and constructor symbols used in the program are given, as well as the type projection functions.

Definition C.1.3 (Γ–Interpretation)

Given an $I\lambda c$–signature

$$\Gamma = \langle \mathcal{V}, \mathcal{C}, r_c, \mathcal{F} \rangle$$

and $\mathcal{P}$ a Γ–program, then a structure

$$\langle E, \mu_0, is_?, v \rangle$$

is called an *interpretation* for $\mathcal{P}$ iff

- $E \in \mathit{events}$ is a set of events,
- $\mu_0 : (\mathcal{F} \cup \mathcal{C}) \to \mathit{value}$ assigns meanings to predefined functions and constructors, and where for all $c \in \mathit{constructorSymbol}$,

  $$r_c\, c = n \;\Rightarrow\; \mu_0\, c = \lambda v_1. \cdots . \lambda v_n . \langle \mathcal{C}[\![c]\!], v_1, \ldots, v_n \rangle$$

- $is_?$ is a pair of relations

  $$is_c, is_t \subseteq \mathit{value}$$

  indicating which values can be used as communication ports and tags,
- $v \in \mathit{value}$ is a value.

and where *value*, *events*, and $\mathcal{C}[\![\,\cdot\,]\!]$ are as defined in section A.2.

□

Intuitively, an interpretation is a model if the interpretation agrees with the semantics of the program as given by the expression function $\mathcal{E}[\![\,\cdot\,]\!]$. This means that some I/O trace must generate the set of events given in the interpretation, and that the function definitions must agree with those of the program.

Definition C.1.4 (Γ–Model)

Given Γ an $I\lambda c$-signature, $\mathcal{P}$ a Γ–program, and $\mathcal{I}$ an interpretation of $\mathcal{P}$, we say $\mathcal{I}$ is a *model* for $\mathcal{P}$, written

$$\mathcal{I} \models \mathcal{P}$$

iff there exists some I/O trace ι such that

$$E = \mathit{eventStream}\ \iota$$

and

$$v = \mathcal{E}[\![\mathcal{P}]\!]\ \mu_0\ \iota$$

If $v \neq \bot$, $\mathcal{I}$ is called a *non-trivial* model for $\mathcal{P}$. In this definition, we use the helper function

$$\mathit{eventStream} \;:\; \mathit{ioTrace} \to \mathit{events}_\bot$$

to extract the I/O events from an I/O trace. This function is defined as:

$$\begin{aligned}
&\mathit{eventStream}\ \langle t, k, c, v, g \rangle \stackrel{\text{def}}{=} \\
&\quad \langle t, k, c, v, g \rangle \\
&\qquad \mathit{if}\ \iota = \langle t, k, c, v, g \rangle,\ \text{for } t \in \mathit{time}, k \in \{\mathrm{i}, \mathrm{o}\}, \\
&\qquad\ \text{and } c, v, g \in \mathit{value}\ s.t.\ is_c(c),\ \text{and } is_t(g),
\end{aligned}$$

$$\begin{aligned}
&\mathit{eventStream}\ \langle \mathrm{r}, n\rangle \stackrel{\text{def}}{=} \emptyset,\ \text{if } n \in I\!N\\
&\mathit{eventStream}\ \langle \mathrm{a}, \iota_1, \iota_2\rangle \stackrel{\text{def}}{=}\\
&\quad E_1 \cup E_2\\
&\quad\ \text{where } E_1 = \mathit{eventStream}(\iota_1), E_2 = \mathit{eventStream}(\iota_2),\\
&\quad\quad \mathit{if}\ E_1 \neq \bot, E_2 \neq \bot,\\
&\quad\quad \text{and for all } \langle t_1, k_1, c_1, v_1, g_1\rangle \in E_1,\\
&\quad\quad\ \text{and} \langle t_2, k_2, c_2, v_2, g_2\rangle \in E_2, t_1 < t_2,\\
&\mathit{eventStream}\ \langle \iota_1, \ldots, \iota_n\rangle \stackrel{\text{def}}{=}\\
&\quad E_1 \cup \cdots \cup E_n\\
&\quad\ \text{where } E_1 = \mathit{eventStream}(\iota_1), \cdots, E_n = \mathit{eventStream}(\iota_n),\\
&\quad\quad \text{and where } E_1 \neq \bot, \cdots, E_n \neq \bot,\\
&\mathit{eventStream}\ \iota \stackrel{\text{def}}{=} \bot,\ \text{otherwise}
\end{aligned}$$

□

C.2 Combined Semantics

Having cast the semantics of $I\lambda c$ programs into a model-theoretic framework, we can now consider the combined semantics of $I\lambda c$ and interaction logic. We first define what it means for two signatures, of the interaction lambda calculus and interaction logic respectively, to be compatible. This means that the function and variable symbols in either signature must be common to both.

Definition C.2.1 (Compatible Signatures)

Let

$$\Gamma = \langle \mathit{variable}, \mathit{constructorSymbol}, r_c, \mathit{functionSymbol}\rangle$$

be an $I\lambda c$–signature, and let

$$\Sigma = \langle \mathit{symbol}, \mathit{variable}, \mathit{cfSymbol}, \mathit{predicateSymbol}, r_p\rangle$$

be a signature of interaction logic, where $\mathit{cfSymbol} = \mathit{constructorSymbol} \cup \mathit{functionSymbol}$. Then Γ and Σ are called *compatible signatures*.

□

Two models for a program and a set of constraints respectively can be combined if the corresponding components of the models are the same: in particular, that the models describe the same set of events.

Definition C.2.2 (Combined Model)

Let Γ be an $I\lambda c$–signature, and let Σ be a compatible signature of interaction logic. Let $\mathcal{P}$ be a Γ–program, and $\mathcal{F}$ a set of Σ–formulae. Then the structure

$$\mathcal{I} = \langle E, \alpha, \phi, \rho_m, is_?, v\rangle$$

is a *model* for the combined program, written

$$\mathcal{I} \models \langle \mathcal{P}, \mathcal{F}\rangle$$

iff $\langle E, \phi, is_?, v\rangle \models \mathcal{P}$ and $\langle \mathit{value}, E, \alpha, \phi, \rho_m, is_?\rangle \models \mathcal{F}$.

If $v \neq \bot$, and $E \neq \emptyset$, we say $\mathcal{I}$ is *non-trivial*.

□

C.3 Transformations

We are ultimately interested in being able to reason about whether two TCFP programs have the same semantics, and in particular, if some function that transforms one TCFP program to another can be expected to always provide the same semantics. This informal notion of "same semantics" is defined to mean that of all the possible behaviours the original program exhibited, the new program must exhibit at least one, and that whereever the old program was defined, the new program must also be defined. That is, transformations are allowed to reduce non-determinism, but must still leave at least one possible execution path. We first define this notion in terms of the semantics of TCFP programs:

Definition C.3.1 (Semantic-Preserving Transformations)

Let Γ and Σ be compatible signatures of $I\lambda c$ and interaction logic respectively. Then any function

$$t : \Gamma\text{–}\mathit{program} \times \Sigma\text{–}\mathit{formulae} \rightarrow \Gamma\text{–}\mathit{program} \times \Sigma\text{–}\mathit{formulae}$$

is called a $\langle\Gamma, \Sigma\rangle$–transformation. We say t is *semantic preserving* iff for all Γ–programs $\mathcal{P}$ and sets of Σ–formulae $\mathcal{F}$ for which there is some non-trivial model, there exists some $\mathcal{I}$ such that

$$\mathcal{I} \models \langle \mathcal{P}, \mathcal{F}\rangle\ \mathit{and}\ \mathcal{I} \models t\ \langle \mathcal{P}, \mathcal{F}\rangle$$

□

While this definition provides us with a precise notion of what it means for a transformation to be semantic-preserving, it is not particularly helpful in *reasoning* about particular transformations. To help us out here, we reformulate the definition in terms of our proof theory, rather than the semantics. Unfortunately, due to the incompleteness of the proof theory, we will not be able to prove that every semantic-preserving transformation is actually semantic-preserving.

Two interesting special cases of transformations are *program* and *constraint* transformations. These cases are functions that transform only the $I\lambda c$ program or the constraints, not both at the same time. In reasoning about transformations, it is convenient to split the transformations into a sequence of program and constraint transformations, rather than trying to reason about both at once.

Definition C.3.2 (Program and Constraint Transformations)

Let Γ and Σ be compatible signatures of $I\lambda c$ and interaction logic respectively, and let t be a $\langle\Gamma, \Sigma\rangle$–transformation. If for all Γ–programs $\mathcal{P}$ and sets of Σ–formulae $\mathcal{F}$, if

$$t\,\langle\mathcal{P}, \mathcal{F}\rangle = \langle\mathcal{P}', \mathcal{F}\rangle$$

for some Γ–program $\mathcal{P}$', then t is called a *program transformation*; if

$$t\,\langle\mathcal{P}, \mathcal{F}\rangle = \langle\mathcal{P}, \mathcal{F}'\rangle$$

for some set of Σ–*formulae* $\mathcal{F}'$, then t is called a *constraint transformation*.

□

We can now formulate how to prove that constraint transformations are semantic-preserving. This is simply a case of proving that all the formulae in the transformed version are a consequence of formulae in the original version.

Theorem C.3.1 (Semantic Preserving Constraint Transformations)

Let Γ and Σ be compatible signatures of $I\lambda c$ and interaction logic respectively, and let t be a constraint transformation over $\langle\Gamma, \Sigma\rangle$. Then t is semantic preserving if for all Γ–programs $\mathcal{P}$ and sets of Σ–formulae $\mathcal{F}$, if

$$t\,\langle\mathcal{P}, \mathcal{F}\rangle = \langle\mathcal{P}', \mathcal{F}\rangle$$

then

$$\mathcal{F} \vdash \mathcal{F}'$$

Proof: Follows directly from theorem B.3.1.
□

Note that because of the incompleteness of the entailment relation "$\vdash$", this theorem does not hold in the other direction.

Bibliography

[1] Martin Abadi. The power of temporal proofs. *Theoretical Computer Science*, 65(1):35–83, 1989.

[2] Apple Computer, Inc. *Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley, 1987.

[3] Apple Computer, Inc. *Inside Macintosh*. Addison-Wesley, 1987.

[4] Apple Computer, Inc. *HyperCard Reference*. Claris Corporation, 1990.

[5] L. Augustsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1987.

[6] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. FormsVBT: A two-view approach to constructing user interfaces. In *CHI'90 Technical Video Program*, volume 53, 1990.

[7] Lee Alton Barford and Bradley T. Vander Zanden. Attribute grammars in constraint-based graphics systems. *Software Practice and Experience*, 19(4):309–328, April 1989.

[8] Brigham Bell, John Rieman, and Clayton Lewis. Usability testing of a graphical programming system: Things we missed in a programming walkthrough. In *ACM SIGCHI 1991*, pages 7–12, April 1991.

[9] Alan Borning. Defining constraints graphically. In *Human Factors in Computing Systems, CHI 1986 Proceedings*, pages 137–143, 1986.

[10] Margaret M. Burnett and Allen L. Ambler. A delcarative approach to event-handling in visual programming languages. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 34–40, 1992.

[11] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association of Computing Machinery*, 24(1):44–67, January 1977.

[12] Luca Cardelli. Building user interfaces by direct manipulation. Technical report, DEC SRC, October 1987.

[13] Magnus Carlsson. *Fudgets – Graphical User Interfaces and I/O in Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1993.

[14] Magnus Carlsson and Thomas Hallgren. Fudgets – a graphical user interface in a functional language. In *Proceedings of Functional Programming Languages and Computer Architectures*, pages 321–330. ACM Press, 1993.

[15] Manuel M.T. Chakravarty and Hendrik C.R. Lock. The implementation of lazy narrowing. In *Proceedings of PLILP, Programming Language Implementation and Logic Programming*, pages 123–134. Springer Verlag, 1991.

[16] Dominique Clément and Janet Incerpi. Specifying the behavior of graphical objects using Esterel. In *Proceedings of TAPSOFT'89*, pages 111–125, 1989.

[17] James R. Cordy and T.C. Nicholas Graham. GVL: A graphical functional language for the specification of output in programming languages. In *Proceedings of the 1990 IEEE International Conference on Computer Languages*, pages 11–22, March 1990.

[18] James R. Cordy and T.C. Nicholas Graham. GVL: Visual specification of graphical output. *Journal of Visual Languages and Computing*, 3:25–47, 1992.

[19] James R. Cordy, Charles Halpern, and Eric Promislow. TXL : A rapid prototyping system for programming language dialects. In *IEEE International Conference on Computer Languages*, pages 280–285, October 1988.

[20] Joelle Coutaz. The construction of user interfaces and the object paradigm. In *Proceedings of ECOOP '87*, pages 121–130, 1987.

[21] Joelle Coutaz. PAC, and object-oriented model for dialog design. In *Proceedings of INTERACT'87*, pages 431–436, 1987.

[22] Joelle Coutaz. Architecture models for interactive software. In *Proceedings of ECOOP '89*, pages 383–399, July 1989.

[23] Herbert Damker. Spezifizierung und Architekturentwurf von Benutzungsschnittstellen: eine Fallstudie in der Clock-Methodologie. Studienarbeit, Universität Karlsruhe, September 1992.

[24] John Darlington and Lyndon While. Controlling the behaviour of functional language systems. In *Proceedings of Functional Programming Languages and Computer Architecture*, volume 274, pages 278–300. Springer Verlag LNCS, 1987.

[25] Michael L. Dertouzos. The user interface *is* the language. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 2, pages 21–30. Jones and Barlett, 1992.

[26] Andrew Dwelly. Functions and dynamic user interfaces. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 371–381. ACM Press, 1989.

[27] Norbert Eisinger and Hans Juergen Ohlbach. Kalküle für die prädikatenlogik erster stufe. In K.H. Bläsius and H.-J. Bürckert, editors, *Deduktions-Systeme*. Oldenbourg Verlag, München, 1987.

[28] Anthony J. Field and Peter G. Harrison, editors. *Functional Programming*. Addison Wesley, 1988.

[29] Bjorn N. Freeman-Benson and Alan Borning. Constraint imperative programming languages for building interactive systems. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 11, pages 161–182. Jones and Barlett, 1992.

[30] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–65, January 1990.

[31] Dov Gabbay. The declarative past and imperative future. In *Temporal Logic in Specification*, volume LNCS 398, pages 409–448. Springer Verlag, 1987.

[32] R. Gabriel. The automatic generation of graphical user interfaces. In *System design: concepts, methods and tools*, pages 330–339. IEEE, April 1988.

[33] R. Gabriel. A formalism for the definition of graphical formulas. In *ACM SIGSMALL symposium on personal computers*, pages 28–36. ACM, May 1988.

[34] Jean Gallier. Constructive logics. part I: A tutorial on proof systems and typed λ-calculi. Technical Report 8, Digital Equipment Corporation Paris Research Laboratory, May 1991.

[35] Jean H. Gallier. *Logic for Computer Science*. John Wiley, New York, 1987.

[36] Anthony Galton, editor. *Temporal Logic and Their Applications*. Academic Press, 1987.

[37] Emden R. Ganser and John H. Reppy. A foundation for user interface construction. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 14, pages 239–260. Jones and Barlett, 1992.

[38] Eric J. Golin. Tool review: Prograph 2.0 from TGS Systems. *Journal of Visual Languages and Computing*, 2:189–194, 1991.

[39] Andrew D. Gordon. An operational semantics for I/O in lazy functional languages. In *Proceedings of Functional Programming Languages and Computer Architectures*, pages 136–145. ACM Press, 1993.

[40] T.C. Nicholas Graham. Constructing user interfaces with functions and temporal constraints. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 16, pages 279–302. Jones and Bartlett, 1992.

[41] T.C. Nicholas Graham. Future research issues in languages for developing user interfaces. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 22, pages 401–418. Jones and Bartlett, 1992.

[42] T.C. Nicholas Graham. Temporal constraint functional programming: a declarative framework for interaction and concurrency. In John Darlington and Roland Dietrich, editors, *Declarative Programming*, pages 83–100. Workshops in Computing, Springer Verlag, April 1992.

[43] T.C. Nicholas Graham and Tore Urnes. Relational views as a model for automatic distributed implementation of multi-user applications. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work (Toronto, Oct. 1992)*, pages 59–66, 1992.

[44] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[45] H. Rex Hartson and Deborah Hix. Human-computer interface development: Concepts and systems. *ACM Computing Surveys*, 21(1):5–92, March 1989.

[46] H. Rex Hartson, Antonio C. Siochi, and Deborah Hix. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, 8(3):181–203, July 1990.

[47] Ralph Hill. Supporting concurrency, communication and synchronization in human-computer interaction: the Sassafrass UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[48] Ralph Hill. The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications. In *ACM SIGCHI 1992*, pages 335–342, April 1992.

[49] Ralph Hill. The *rendezvous* constraint maintenance system. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 225–234, 1993.

[50] Ralph D. Hill. Languages for construction of multi-user multi-media synchronous (MUMMS) applications. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, pages 125–146. Jones and Bartlett, 1992.

[51] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[52] Bruce Horn. Constraint patterns as a basis for object-oriented programming. In *Proceedings of OOPSLA'92*, pages 218–234. ACM Press, 1992.

[53] Bruce Horn. Properties of user interface systems and the siri programming language. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, chapter 13, pages 211–238. Jones and Barlett, 1992.

[54] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[55] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/RR-665, Yale University, March 1989.

[56] Paul Hudak and Philip Wadler. Report on the functional programming language Haskell (v1.1). Technical Report YALEU/DCS/RR777, Yale University, August 1991.

[57] Kent Karlsson. Nebula – a functional operating system. Technical report, Chalmers University, 1981.

[58] Michael J. Knister and Atul Prakash. Distedit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work (Los Angeles, Ca., Oct. 7–10)*, pages 343–355, New York, 1990. ACM Press.

[59] Glen E. Krasner and Stephen T. Pope. A cookbook for the using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[60] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–184, 1992.

[61] TGS Systems Limited. *Prograph Reference*. TGS Systems, 1989.

[62] Mark A. Linton, John M. Vlissides, and Paul R Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[63] Hendrik C.R. Lock. An amalgamation of functional and logic programming languages. Technical Report 408, GMD, September 1989.

[64] S. Matwin and T. Pietrzykowski. Prograph: A preliminary report. *Computer Languages*, 10(2):91–126, 1985.

[65] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical Report ECS-LFCS-89-85, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[66] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Hemel Hempstead, 1986.

[67] Catherine Morton. Tool support for component-based programming. Master's thesis, York University, North York, Canada, June 1994.

[68] Brad A. Myers. Creating dynamic interaction techniques by demonstration. In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, pages 271–278. ACM Press, November 1987.

[69] Brad A. Myers. Creating user interfaces by demonstration. Technical Report CSRI-196, Computer Systems Research Institute, May 1987.

[70] Brad A. Myers. A new model for handling input. *Transactions on Information Systems*, pages 289–320, July 1990.

[71] Brad A. Myers. Separating application code from toolkits: Eliminating the spaghetti code of callbacks. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 211–220. ACM, acm press, November 1991.

[72] Brad A. Myers. Ideas from Garnet for future user interface programming languages. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, pages 147–160. Jones and Bartlett, 1992.

[73] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Computer Science Department, Carnegie Mellon University, July 1993.

[74] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[75] Brad A. Myers, Dario A. Guise, and Brad Vander Zanden. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. In *OOPSLA'92, ACM SIGPLAN Notices, Vol. 27, Num. 10, Oct. 1992*, 1992.

[76] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Human Factors in Computing Systems*, pages 195–202, Monterrey, CA, May 1992.

[77] Roy Nejabi. Efficient semi-replicated implementation of multi-user user interfaces. Master's thesis, York University, North York, Canada, May 1995. (expected).

[78] NeXT Inc. *NeXTStep Reference*. Addison-Wesley, 1991.

[79] Rob Noble and Colin Runciman. Functional languages and graphical user interfaces – a review and a case study. Technical Report YCS-94-223, University of York, Febraury 1994.

[80] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *Proceedings of PARLE '91*, volume LNCS 506, pages 202–219. Springer Verlag, June 1991.

[81] Adrian Nye. *Xlib Programming Manual*. O'Reilly and Associates, 1988.

[82] Dan R. Olsen Jr. Propositional production systems for dialog specification. In *ACM CHI '90 Conference Proceedings*, pages 57–64, 1990.

[83] Dan R. Olsen, Jr. *User Interface Management Systems: Models and Algorithms*. Morgan Kaufmann, San Mateo, 1992.

[84] D.R. Olsen Jr. and E.P. Dempsey. SYNGRAPH: A graphical user interface generator. *Computer Graphics*, 17(3):43–50, 1983.

[85] Open Software Foundation. *OSF/Motif Style Guide*. Prentice Hall International, 1990.

[86] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the 1991 USENIX Winter Conference*, pages 105–115, January 1991.

[87] John F. Patterson, Ralph D. Hill, Steven L. Rohall, and W. Scott Meeks. Rendezvous: An architecture for synchronous multi-user applications. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 317–328. ACM, October 1990.

[88] F. Newberry Paulisch and W.F. Tichy. EDGE: An extensible graph editor. *Software Practice and Experience*, 20(S1):63–88, 1990.

[89] Randy Pausch, Nathaniel R. Young II, and Robert DeLine. SUIT: the Pascal of user interface toolkits. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 117–125. ACM, acm press, November 1991.

[90] Peter Pepper. The programming language OPAL-1. Technical Report May-91, T.U. Berlin, May 1991.

[91] Nigel Perry. I/O and inter-language calling for functional languages. In *Proceedings of the Ninth International Conference of the Chilean Computer Science Society and Fifteenth Latin American Conference on Informatics*, July 1989.

[92] Nigel Perry. Towards a concurrent object/process oriented functional language. In *Proceedings of ACSC'15, the Australian Computer Science Conference*, 1992.

[93] Guenther E. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag, Berlin, November 1983.

[94] T. W. Reps and T. Teitelbaum. *The synthesizer generator*. Springer Verlag, 1989.

[95] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[96] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, Dubuque, Iowa, 1986.

[97] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[98] Duncan C. Sinclair. Graphical user interfaces for Haskell. In *Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming*, pages 252–257, July 1992.

[99] Gurminder Singh and Mark Green. Automating the lexical and syntactic design of graphical user interfaces: The UofA* UIMS. *ACM Transactions on Graphics*, pages 213–254, July 1991.

[100] David Canfield Smith and Joshua Susser. A component architecture for personal computer software. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, pages 31–56. Jones and Bartlett, 1992.

[101] Gekun Song. Mixing visual and textual programming in functional languages. Master's thesis, York University, North York, Canada, February 1995. (expected).

[102] Mark Stefik, Gregg Foster, Daniel G. Borrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[103] William Stoye. A new scheme for writing functional operating systems. Technical Report 56, Cambridge University, 1984.

[104] Sun Microsystems Inc. *OPEN LOOK Graphical User Interface Style Guidelines*. Addison-Wesley, 1990.

[105] Shin Takahashi, Satoshi Matsuoka, Akinora Yonezawa, and Tomihisa Kamada. A general framework for bi-directional translation between abstract and pictoral data. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 165–174, November 1991.

[106] Leo Tessler. Keynote speech. In *CHI 89: Conference on Human Factors in Computing Systems*. Apple Computer, Inc., May 1989.

[107] Linda Tetzlaff and David R. Schwartz. The use of guidelines in user interface design. In *Human Factors in Computing Systems, CHI 1991 Proceedings*, pages 329–333, April 1991.

[108] Simon Thompson. Interactive functional programs: a method and a formal semantics. Technical Report UKC-48, University of Kent at Canterbury, November 1987.

[109] Tore Urnes. A relational model for programming concurrent and distributed user interfaces. Master's thesis, Norwegian Institute of Technology, University of Trondheim, 1992.

[110] Marko van Eekelen, Halbe Huitema, Eric Nöker, Sjaak Smetsers, and Rinus Plasmeijer. Concurrent Clean language manual (version 0.8.4). Distributed with Concurrent Clean implementation, February 1993.

[111] Bradley T. Vander Zanden. An active-value spreadsheet model for interactive languages. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, pages 183–210. Jones and Bartlett, 1992.

[112] Bradley T. Vander Zanden, Brad A. Myers, Dario Giuse, and Pedro Szekely. The importance of pointer variables in constraint models. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 155–164. ACM, acm press, November 1991.

[113] Philip Wadler. Comprehending Monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, June 1990.

[114] Philip Wadler. The essence of functional programming. In *19th Annual Symposium on Principles of Programming Languages*, pages 1–14, Santa Fe, New Mexico, January 1992.