A compositional model for the formal specification of user interface software

Panagiotis Markopoulos

Submitted for the degree of Doctor of Philosophy
March 1997

A compositional model for the formal specification of user interface software

Panos Markopoulos

Submitted for the degree of Doctor of Philosophy, March 1997.
Queen Mary and Westfield College
University of London.

Abstract

This thesis investigates abstractions for modelling user interface software, discussing their content and their formal representation. Specifically, it focuses on a class of models, called formal interactor models, that incorporate some of the structure of the user interface software. One such abstract model is put forward. This model is called the Abstraction-Display-Controller (ADC) interactor model; its definition draws from research into user interface architectures and from earlier approaches to the formal specification of user interfaces.

The ADC formal interactor model is specified using a specification language called LOTOS. Several small scale examples and a sizeable specification case study demonstrate its use. A more rigorous discussion of the ADC model documents its properties as a representation scheme. The ADC interactor is compositional, meaning that as a concept and as a representation scheme it applies both to the user interface as a whole and also to its components. This property is preserved when interactors are combined to describe more complex entities or, conversely, when an interactor is decomposed into smaller scale interactors. The compositionality property is formulated in terms of some theorems which are proven. A discussion on the uses of the ADC model shows that it provides a framework for integrating existing research results in the verification of formally specified user interface software. Finally, the thesis proposes a conceptual and formal framework for relating interface models to models of users' task knowledge capturing some intuitions underlying task based design approaches.

Acknowledgements

I wish to acknowledge the help of all my colleagues in the Department of Computer Science at Queen Mary and Westfield College, who provided an active research environment, practical support and advice. In particular, I would like to thank Peter Johnson and Jon Rowson for their supervision.

I am grateful to Stephanie Wilson and John Samuel for all their work in proof reading the thesis, but also for advising and encouraging me while I was working on it. Also, I would like to thank Eamonn O'Neill for helping debug parts of the thesis.

Acknowledgement is also due to INRIA (France), to UPM (Madrid) and CNR (Pisa) for providing the software tools which I have used in this research.

Finally, I owe a big thank you to my parents Dimitris and Christina for caring and for supporting me morally and materially throughout my studies, and to my wife Annick who endured the long build-up to the submission of the thesis and made it a happy time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the main themes of the thesis: Human-Computer Interaction, Formal Methods in Software Engineering and, more specifically, the application of Formal Methods to the design and development of user interface software. The user interface is a component of the software and hardware implementing an interactive system, which mediates between the computer and its human user. Methods and tools are needed to support the engineering of user interfaces [15, 87] and Formal Methods are discussed in this light. This chapter raises the central question: "What are the appropriate abstractions for modelling and engineering user interface software?" The thesis discusses properties of such abstractions and proposes an appropriate formal model. This chapter concludes with an overview of the thesis.

1.1 Human Computer Interaction

Human-Computer Interaction (HCI) is a multi disciplinary field that concerns the use of interactive computer systems by humans. Diverse scientific disciplines, such as psychology, ergonomics and sociology, may help provide theories and models for the explanation of the phenomena pertaining to the interaction of humans and computers. Graphic and industrial design may help improve the presentation of interactive systems. Computer science and engineering are necessary to build these systems.

The central problem for HCI is an engineering one, that of developing systems that will satisfy the needs of their users. Arguably, the major portion of this engineering effort is in developing the software for the user interface [139]. The user interface mediates between two participants in interaction: a human operator and the computer hardware and software that implement the interactive system. The development of the user interface is inherently difficult [138]. Current practice in user interface design relies largely upon the talent of individual designers and in this sense it may be considered as a craft, if not an art [51]. Much is to be gained if the study of human-computer interaction

can provide guidance in terms of engineering principles, tools and methods specific to the problem of designing and developing user interface software.

The research presented in this thesis adopts this software engineering view of HCI. This should not suggest that the importance of the other contributing disciplines is not recognised. On the contrary, it is a definitive aim of research in software engineering methods in HCI to provide the practical means by which such bodies of knowledge may feed into the development of computer systems. This thesis does not provide a theory of what is a usable system or of how the productivity and the creativity of the user of a system may be assisted. It is concerned with the development of theoretical models that describe human-computer interfaces and the study of their use so that they assist the engineering of user interfaces. In particular, it examines how a class of software engineering methods, known as formal methods, can contribute to this task.

1.2 Formal methods of software engineering

Formal methods are mathematically based techniques used in system development. They provide a framework for systematically specifying, developing, and analysing systems. A formal method is associated with a formal specification language, which has a formal syntax and a formal semantics, and is based upon a logical inference system [190]. The latter is a way of characterising which objects in the semantic domain satisfy a specification. It is important to maintain the distinction between the specification and the object it describes. The term *specificand* [190] is used to refer to the latter. In this thesis the specificand is the user interface software or some part of it. A specification is a model of the specificand. In other words, the specification can be seen as a representation from which predictions are derived regarding the object it represents. Predictions may be made directly using the logical inference system that underlies the formal method or indirectly via tool support that is based on this inference system.

Formal specifications are an attractive representation technique for the software engineer because they are economical and they are unambiguous. They are economical because they can abstract away from detail that is not relevant for the predictions they aim to make [38]. By means of abstraction a formal specification describes a class of objects that adhere to the same abstract description. There may be numerous abstractions of the same specificand, e.g. in terms of rules pertaining to its behaviour, the timing of its behaviour, or even stochastic aspects [176]. A specification may address various *abstraction levels,* e.g. for graphical output they may concern pixels, windows, etc. A formal specification is unambiguous because it has only one meaning: it should always be clear whether a particular object of the semantic domain belongs to the class of objects described by the specification. This is the main difference from informal specifications which different readers may interpret differently or even the same person might interpret in different ways at different times. Finally, a minimal requirement from any formal specification is that it be *consistent*, in that no two contradictory statements can be inferred from the specification.

But when is a formal specification a useful engineering tool? Clearly, the set of predictions made with the formal method should be relevant to the task for which they are used. To apply formal methods to a particular problem domain, the general characteristics of the objects of this domain have to be captured in a generic model [192]. The success of such a model depends on its potential to produce valid and useful predictions about the problems of the particular domain. The validity of the model can be refuted but it cannot be proved [38]. However, confidence that the model is an appropriate representation for the objects of the problem domain that it represents increases with each successful application of the model. The practical application of the model also seems to be the only definitive testimony of how useful it may be. As Gaudel [73] points out, a formal model tailored for a specific problem domain should mask, as far as possible, the underlying mathematical concepts. This will make it more accessible to the persons involved in the software development.

Another important issue in the application of formal methods is the need to structure specifications in order to master their size and complexity. Finding an appropriate structure for formal specifications is a key problem for scaling up the use of formal methods to industrial scale problems [179]. Methods for structuring specifications are dependent on the application area, so this thesis will examine such methods in the context of specifying user interface software.

Another dimension for characterising a formal model is its expressive power, i.e. the range of the objects in the represented world that are successfully described by the model. The quality of the predictions made depends on the analytical power of the model. In turn, this depends on the underlying specification language and its expressive power. Usually, the weaker the expressive power, the stronger the analytical power will be [38]. This suggests that special purpose models of interface software that have a modest expressive power may be preferred over general abstractions or more powerful formal models.

The specification is subject to several manipulations that are used in the development process. These manipulations, or transformations, are essential tools in the development of any system [38]. They must be guaranteed to be property-preserving. For example, a specification may be broken down to smaller modules, it may be used to construct higher level specifications, or it may be refined to a more detailed description of a behaviour that satisfies it [158]. The specification may be used as the basis for testing an implemented system or for verifying a system formally.

The real test for formal methods is the pragmatic issue of whether they aid system development. The utility and the acceptance of a formal method depends on the efficiency with which it is applied, how its use affects the quality of the software produced, and the potential of scaling up its use to problems of realistic size. It is an old and on-going debate whether or not formal methods benefit software development. Some of the most controversial issues are discussed in [24, 80]. The role that formal methods should play in software development is an issue of a strong debate even among some of their most staunch supporters. The reader is referred to [166] for a discussion

on the role of formal methods, by some of their most outstanding proponents and sceptics.

Currently no scientific assessment has been undertaken, concerning the impact formal methods have on the processes and products of the computer industry [26, 74]. The state of the technology in the area is changing constantly and a verdict on the benefits of formal methods, in a particular problem domain, can only be ephemeral. It seems a more pragmatic and fruitful research venture to try to understand the limitations of current methods, with respect to a particular problem domain, and to investigate how improvements can be achieved.

1.3 The application of formal methods to HCI

It is a widespread view in the software engineering community that formal methods are not applicable in the development of user interfaces, e.g. [25]. However, the application of formal methods to the study of HCI has a history of more than ten years. Formal languages, i.e. languages with a formal syntax, have been adopted by HCI researchers as notations to specify and communicate models of users, systems and interaction, e.g. [78, 86]. A strand of research closer to the traditions of software engineering has been concerned with developing abstract models of interactive systems, which are used to predict the usability consequences of design decisions. This has resulted in the generic mathematical formulation of properties that are related to the usability of a specified system [84, 85, 49]. The verification of interaction properties using formal models may provide greater assurance of software usability earlier in the design process.

Research into abstract models of interactive systems can been criticised on, at least, two accounts. One is the validity of the predictions, which is an empirical question [49, pp. 77]. Another is the difficulty of applying abstract models to the construction and concrete detail which reflects how the system is actually built [49, pp. 336]. This thesis focuses on a class of formal models that aim to improve on the latter account. These models incorporate characteristics particular to the software architecture of user interfaces, so they are referred to as *formal architectural models*. Numerous formal architectural models have been proposed. Recently, the term *interactor models* has been adopted to refer to them collectively, e.g. [54, 62, 150]. The term is given a concrete definition in later chapters. Interactor models are very similar to their predecessors, the abstract models of user interfaces. An argument put forward in this thesis is that their evolution should reflect on developments in less formally defined models of the interface software architecture.

Existing interactor models have diverse origins and are defined using various formal specification languages. The diversity between formal interactor models can be traced back to their different purposes, different formal frameworks, and different historical origins. However, they demonstrate a strong convergence in their essential elements [54]. At the very least, this convergence shows a consensus among researchers, concerning the essential elements of interface software that should be captured by an

architectural model. This search for the 'right abstraction' is characteristic of research into applying formal methods to the development of human computer interfaces.

1.4 The thesis and the research method

The thesis put forward is that the practical application of formal methods to the design and development of user interface software can be assisted by a class of formal models called interactor models. In particular, the thesis proposes an interactor model which is called Abstraction-Display-Controller (ADC) after its main components. It is suggested that this model is an appropriate abstraction for modelling user interface software because it captures essential and general characteristics of user interface software and it maintains a close relation to software architectures. As a formal model the ADC interactor model is equipped with some properties that are essential for its use as a design representation in the development and analysis of a user interface. Finally, it is suggested that it provides a conceptual and formal framework for integrating diverse approaches to modelling and analysing user interface software.

Considering the multi disciplinary nature of HCI, this thesis is biased towards computer science. This has its consequences for the nature of the research reported. The requirements set for the ADC interactor model have emerged from a comparison of user interface architectures and existing formal models. The formal method used for its specification was adopted after assessing earlier research approaches in this field. Formal methods differ with respect to their applicability for specifying user interface software, the available tool support, and how widespread their use may be. The thesis discusses the choice among the candidate formal methods and the trade-off between the expressive power of the notation and the feasibility of manipulating and using the formal specifications in practice.

An important consideration throughout is to facilitate the practical application of formal methods as aids in the design and development of user interface software. The thesis argues that for an interactor model to be useful as an engineering and conceptual tool it should be possible to specify the user interface as a composition of interactors. The compositions formed ought to be interactors themselves. This property, called the compositionality of the model, is consistent with informal abstractions of user interface software, and enables the recursive application of the model at various abstraction levels. Conversely it should be possible to decompose interactors into smaller scale interactors. Further, it should be possible to infer salient properties of a designed interface from its specification. In the body of this thesis these propositions are discussed in a formal framework and the ADC model is defined to support them.

The ADC model was tested with a reverse-engineered formal specification of a simple, but real, multi media application. This case study prompted improvements to the model, which are discussed. Further, the case study is testimony to the validity of the model. Some of the engineering properties that were set as requirements for the model have been proved as theorems. On the basis of these theorems, formal transformations of the

interactor specifications are defined. Where necessary an algorithmic solution to the transformation problem is proposed. The analytical use of the model is discussed. The ADC model provides a formal and conceptual framework that enables the integration of earlier results regarding the specification and verification of usability related properties of a user interface.

The contribution of the thesis is two-fold. From a computer science point of view it shows the application of formal methods to a broad and interesting problem domain, namely, the design and development of user interface software. Within this particular problem domain, the ADC interactor model can help one to write and to understand complex formal specifications. For example, the case study discussed is an interesting problem simply from a specification point of view. The application was modelled to such detail and complexity that the specification tested the limits of the tool support used. The ADC model provides a framework for identifying and combining reusable specification components for user interface software.

More importantly, a formal model, such as the ADC interactor, provides a bridge between the world of the mathematical concepts of formal methods and the domain of user interface software. As has been argued in section 1.3, this is a necessary prerequisite for applying formal models to any application domain. A formal architectural model like ADC fills the gap between more abstract models of interface software and the architectural design of actual software. From an HCI point of view, the thesis aspires to provide the interface designers or developers with a formal modelling scheme appropriate for their tasks. The model embodies essential elements of interface software architectures, it can help establish usability related properties and the correctness of the interface software. Finally, it contributes towards a long-standing aim of research in applying formal methods in HCI. This aim is to provide a representation scheme that helps express and apply requirements derived from a human-centred study of the work supported by the computer. This scheme, it is argued, is neutral with respect to theories of human cognition and behaviour, but can provide the expressive means for relating such theories to representations of user interface design.

1.5 Overview of the thesis

This chapter has posed a two-fold question that this thesis tries to answer: ‘What are the right abstractions for modelling user interface software?’ and, ‘what is a suitable representation scheme for them?’. In the body of this thesis, the ADC interactor model is put forward as the appropriate answer to these questions, and its formal specification is discussed. A substantial part of the thesis is concerned with documenting the properties and demonstrating the use of this model. Chapters 2 and 3 prepare the ground for the introduction of the ADC model, surveying two different streams of research. These reflect the diverse influences on the development of the ADC interactor model.

The quest for the ‘right abstraction’ has much in common with the endeavours of research into user interface software architectures. Chapter 2 reviews this research.

Software architectures are assessed in terms of the practical benefits they offer to the user interface developer. The aim of this review is to identify, and in subsequent chapters to inject into the formal model, some of the practical merits of software architectures and some of their morphological characteristics. This approach should make the model more relevant to the problems of the user interface designer and a more useful engineering tool. The review of software architectures for user interfaces in chapter 2 plays a bootstrapping role for the thesis. It summarises some standard terminology and it defines user interface software, its role, scope, and ontology. It offers an insight into the nature of the specificand, the user interface software, which is the first step in the application of a formal method.

The second major influence on the development of the ADC interactor model is the research tradition in formal methods in HCI. Chapter 3 presents a very selective view of the developments in the field, in order to demonstrate the current trend towards interactor models. It first discusses the ad hoc application of general purpose formal specification languages to specify user interfaces. Then it discusses *abstract models* of interaction. Abstract models are a class of models that afford expressions of properties of interactive systems with minimal reference to their internal structure. These properties capture design intuitions that pertain to the usability of an interactive system. An important motivation throughout the development of the ADC model was to attain formulations of these properties at a more concrete level of description. In chapter 3 those elements that make interactors a valid abstraction for interface software are highlighted. Further, their qualities as engineering tools are discussed. The derived requirements prompt the definition of the interactor model of the following chapter. Finally, chapter 3 discusses LOTOS [100] which is the formal method adopted for the specification of user interfaces in the thesis. This choice involves some conscious trade-offs which are flagged.

Chapter 4 introduces the ADC interactor model, informally first by defining its components and their properties, and then as a formal specification template in the LOTOS language. Some simple examples are discussed. The ADC model is compared briefly with other architectural models both formal and informal.

Chapter 5 presents the application of the ADC model to specify a multimedia application for the Macintosh computer. The specification is quite large and, therefore, it represents a real test for the modelling scheme and the tool support used. This case study resulted in several developments to the ADC model which are summarised at the end of the chapter.

Chapter 6 revolves around the theme of the *compositionality* of the ADC interactor model. In this thesis, the term compositionality denotes the ability to compose instances of a model to obtain a new instance of the model. Further, this instance of the model should be used for further manipulations with minimal reference to its constituent parts [38, 73]. Chapter 6 gives a rigorous definition of the requirement for compositionality. In order to show that the ADC model does indeed posses this property, the model is defined in a manner more rigorous than in chapter 4. This enables the definition of two

syntactical transformations of interface specifications: the synthesis and the decomposition of ADC interactors. For the first, and, where possible, for the second, the meaning of the specification is preserved up to the strongest possible level, that of *strong bisimulation equivalence* [133]. The synthesis transformation has been presented partly in [125] and an early discussion of the decomposition transformation has been outlined in [126]. These concepts are discussed thoroughly with their theoretical foundation and a reflection on their practical implications.

Chapter 7 discusses the use of the ADC model. Formulations of properties related to the usability of a system previously expressed in terms of abstract models, are discussed in the framework of the ADC model, updating an early discussion on the topic in [123]. The verification of properties of the *dialogue* supported by an interface is examined. Dialogue pertains to the syntactic regularities of the temporal structure of the interaction between human and computer. Its study has been a traditional concern for human computer interface design notations, e.g. [77, 88]. On a practical note, chapter 7 discusses the automatic verification of user interface properties, reflecting on the currently available tool support and its limitations in supporting the analytical use of the ADC model. In the latter half of chapter 7 the discussion extends to some broader topics. The use of the ADC model within a traditional top-down software engineering approach is demonstrated and, also, its use in the context of a more psychologically informed approach to design. Note that the historical origins of the ADC model have been an attempt to support a 'model based' design of user interfaces with formal specification techniques, see [187]. The ADC model is discussed in conjunction with a formal representation of user tasks. This formalisation gives a novel insight into task based design. Further chapter 7 proposes a framework that helps relate the formal representations discussed in the thesis to more informal and practical techniques for the design and evaluation of human computer interfaces.

The final chapter summarises the thesis, it provides an assessment of its contribution and it discusses a series of emerging issues for future research.

Chapter 2

User interface architectures

This chapter defines an ontology and a terminology, regarding user interface software, which is adopted in later chapters. As far as possible, the commonly accepted definitions are used in the form that they result from international workshops and reference models. Some definitions may seem contrived but they are necessary to discuss concisely and unambiguously user interfaces in later chapters. The emphasis is on defining the notion of an architecture and on presenting some of the most influential proposals. The trends which are observed in the evolution of user interface architectures influence the definition of the formal model in later chapters.

2.1 Some basic terminology

The terminology introduced in this section has been proposed by the IFIP working group 2.7 on user interface engineering [14, 181]. The term *interactive system* is taken to refer to the whole system developed for end-users. Alternatively, the term *application* is used instead with the same meaning. An interactive system is composed of two components, *the functional core* and the *user interface system*.

An interactive system is associated with a particular problem domain, in which it is intended to be used, e.g. controlling some automated process, writing documents, storing and retrieving information, supporting communication, etc. The functional core (FC) implements domain dependent concepts. The user interface system (UIS) implements the interaction between a user and the functional core. The term user interface, or simply interface, is interchangeable in this thesis with the term UIS.

There are a number of tools that may support the developer in building the user interface system. They are characterised below in terms of their functionality.

- A collection of objects that implement user input and presentation of information constitutes an *interaction toolkit*.

- An ensemble of tools and software for specifying, building, and evaluating the user interface constitutes a *user interface development environment* (UIDE).
- A *user interface development tool* (UIDT) is a specialised tool that handles only particular aspects of interface design.
- A *user interface runtime system* (UIRS) is the run-time environment that supports the user interface.
- A *user interface management system* (UIMS) is the assemblage of UIDE, UIDT and UIRS. For many authors, the term UIMS has a narrower definition, including the UIRS and some UIDTs only.

A *property* of an interactive system is a feature describing some aspect of the system. It may be used as a criterion for the evaluation of the system. However, a property is neither good or bad; it partially defines a design space. For example, *flexibility* is a property of an interactive system that supports many alternative ways of performing the same task. A property becomes a *requirement* once its satisfaction is considered significant for a design. A requirement which must be fully satisfied by the design and/or the implementation of the system is a *constraint*. A *principle* is a set of properties which should be satisfied by all designs or a major class of designs.

2.2 Architectures of user-interface systems

The term architecture is widely used in software engineering. It applies to all types software systems. Examples of architectures are the viewing pipeline for computer graphics, the decomposition of compilers to lexical analysers, parsers, code analysers and code generators, etc. In this thesis, the term will be taken to refer to user-interface systems unless otherwise qualified. The term is not used consistently in the literature so a brief clarification is helpful.

An architecture of a system identifies its components and their inter-component interfaces [37]. It is an abstraction of a set of concrete implementations. In many cases a UIMS may enforce a particular architecture for a UIS. Otherwise it may be enforced ‘manually’ by programming. An abundance of architectures was reported in the 1980s but their enumeration is not of interest in the present discussion. However, architectures are important. They embody design knowledge pertaining to the structure of an interactive system. An architecture should, ideally, ‘package’ this knowledge and enable its re-use.

An architecture may define a particular configuration for the connection of software components, e.g. they may be organised in a hierarchy, a pipeline, a graph, etc. It may go further and define a particular way of composing or decomposing its elements, e.g. top-down or bottom-up, in which case it affects the design strategy [37]. As architectures become more concrete, they identify re-usable components and

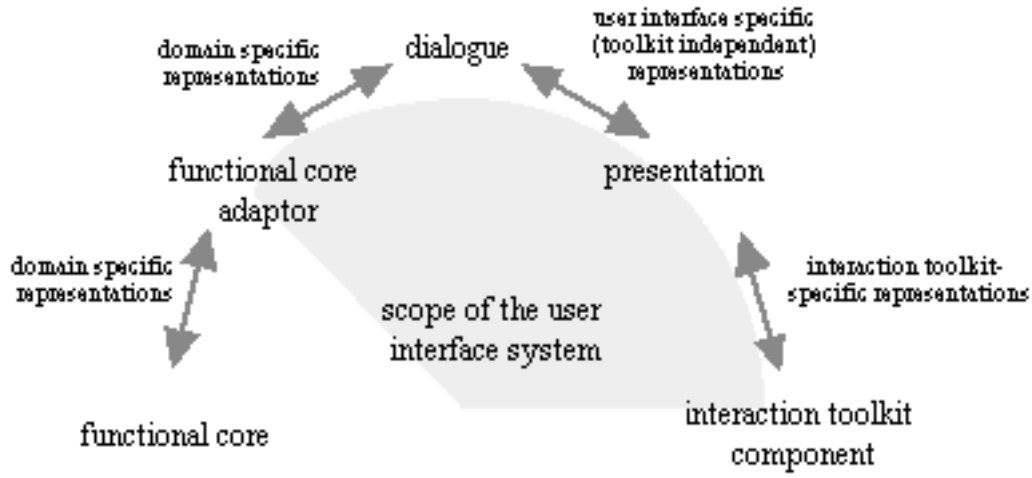


Figure 2.1. The Arch reference model: its components and data representations. Arrows indicate data flow. The shaded arch illustrates the scope of the user interface system.

mechanisms for the communication of data and control, e.g. constraint systems, procedure calls, message passing, etc.

Architectural models are abstract representations of architectures. They are generic templates which may be instantiated by concrete architectures. In many cases, this instantiation process is totally informal and the architectural models are quite simply heuristics or conceptual aids for the design of an architecture or directly for the design of a user interface system. Such architectural models are sometimes called conceptual software architectures.

2.3 A reference model for user interface architectures

A reference model is a conceptual architectural model, which is not intended to be used generatively by its instantiation to a particular system, but only descriptively for categorising the components of an interactive system and discussing its architecture [14]. A series of international workshops have attempted to define commonly accepted user interface architectures. These were not always intended as reference models, but they have had a bigger impact as such. The most influential has been the Seeheim workshop, summarised in [77]. More recently the Arch model [14, 181] was proposed as a modification of the Seeheim model. This thesis adopts the Arch model for reference purposes.

The Arch model has five layers, shown schematically in figure 2.1. It is shaped and named as an arch, to indicate that in most cases its two extremes are constraints upon the design. The layers, or components, are defined below as in [14].

- The *functional core* controls, manipulates, and retrieves domain data and performs domain-related functions.
- The *interaction toolkit* implements the physical interaction with the end-user.
- The *dialogue component* is responsible for task-level sequencing, for the user and for the portion of the functional core that depends upon the user, e.g. for providing multiple view consistency, and for mapping back and forth between domain-specific formalisms and user interface specific representations of data.
- The *presentation component* mediates between the dialogue and the interaction-toolkit components. It provides the dialogue component with toolkit-independent data representations.
- The *functional core adaptor* component mediates between the dialogue and the functional core. It implements functionality necessary for the human operation of the system that is not available in the functional core.

The Arch model identifies three types of data representations used in an interactive system. Representations of the *domain data*, e.g. text for a word processing facility, are assumed to be independent of the interface. Representations of the media used for input and output, e.g. pixels, postscript, which are called *interaction toolkit dependent representations*. There may also be intermediate *user interface specific representations*, which are independent of the interaction toolkit, e.g. paragraph styles, column information, etc. Data flows bi-directionally along the arch and is transformed from one representation to another.

In different application domains, the emphasis may shift between the components of the Arch. This is demonstrated by the Slinky model [181] which in essence is identical to Arch. The Slinky model uses the analogy of a toy, called Slinky. The toy has a very loose spring with many coils that can ‘walk’ down stairs by shifting its mass along its arch-shape. The analogy indicates that any of the components of the Arch may concentrate a larger portion of the functionality of the interactive system. The functionality distribution changes depending upon the domain and the particular system design. Slinky has been called a meta-model to differentiate it from the Arch model [181]. Given the authors’ own definition of a reference model, mentioned previously, this refinement to the Arch model is unnecessary and possibly misleading, as it is not the purpose of a reference model to prescribe the boundaries of its components and the allocation of functions to them.

2.4 Separation of interface and application

In the basic terminology, the user interface system and the functional core are defined as distinct entities. This is a cornerstone assumption for most user interface architectures [191], and the Arch model [14] has inherited it from its predecessor, the Seeheim model [77]. From a software engineering point of view, separation may support the modularity

of interface software, multiple interfaces for one application and the consequent customisation of the interface.

Not all software developers, or HCI researchers, agree on the utility of separation. In fact, the desired degree of separation has been shown to depend on the design problem. Opinions on its existence and utility are largely affected by the design process and organisational factors [162]. Rosson et al. [162] distinguish several types of separation. Separation may apply only at the level of the conceptual architecture, in which case it may be called *conceptual separation*. Separation is enhanced when it carries through to the development of the interface software and is enforced in the resulting implementation architecture of the system. This latter form of separation has been called *architectural* or *implementational separation.*

Implementational separation is supported by the functional core adaptor component which explicitly specifies the communication between the UIS and the functional core. The communication may be controlled *internally* by the functional core, *externally* by the interface or the control might be *shared* [88]. With internal control, the functional core calls interface procedures when it requires some input or output to be performed. With external control, the UIS manages input and output and notifies the functional core when necessary. Internal control makes it harder to implement multiple threads of activity, as complex dialogue structures may need to be encoded in the functional core. Most UIS architectures in the discussion that follows assume that control is external or shared.

The diversity of approaches outlined suggests that a generic interface architecture should not enforce a particular scheme for the separability of the user interface. Separability should be supported where necessary and any scheme for communication control should be within the range of the model. In terms of the Arch reference model, the extent of the user interface system and the relative roles of the components vary with the context of the application of an architectural model.

2.5 Object-based architectures

The Arch model, as well as its predecessor the Seeheim model, describe interface components at a high level of abstraction. They do not detail the structure of these components. The reference model is not intended to be used as a concrete architecture. A layered implementation architecture, whose layers would correspond to the components of the Arch model, would be inefficient. The main reason for this is that semantic information of the functional core is needed frequently at the presentation component, in particular for direct manipulation systems. This inflicts communication overheads for a layered implementation.

This observation has led to the development of object based architectures. The idea behind these is that the interface can be constructed as collection of objects [41]. The interface objects exist in a *homogeneous object space* [41], i.e. interface objects do not

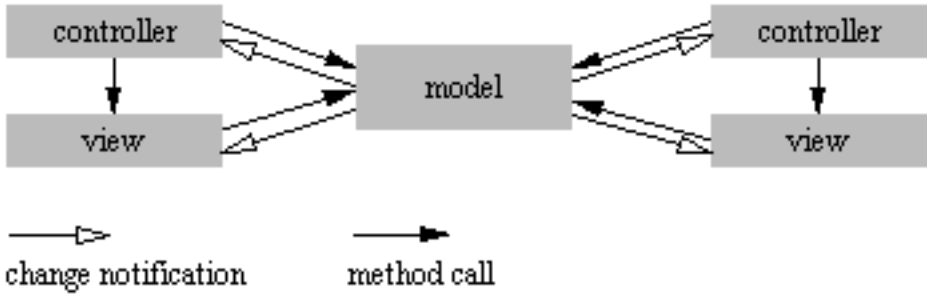


Figure 2.2. The MVC architecture. A model and two view-controller pairs.

belong to layers managing different representations and offering a common function, e.g. in accordance with the Arch reference model. Each object may transcend several of the components of the reference model. An object-based architectural model explicates the common structure of all objects and how objects are related to each other. This concept has become popular because of its potential to support efficient feedback, concurrency, distribution, multiple threads of input and modular development.

Some influential object-based approaches are discussed in the remainder of this chapter. At times, these approaches use the terms ‘agents’ or ‘interactors’ to refer to the abstraction units they deal with. The term ‘object’ is used here, as in object-oriented software engineering, to describe an entity with some private state and processing ability which is accessed by its environment only through a fixed ‘interface’. Some of the approaches that are discussed in this thesis have used the term ‘agent’ [1, 39, 15] to qualify the objects as active entities, as opposed to a passive manipulable information store. They use the term ‘interactor’ to describe agents, or objects, that are displayed to the user. When not referring to specific models and their own terminology, this thesis distinguishes interactors as special cases of objects that have a display function and, in general, support both input and output.

2.5.1 The MVC architecture

The model-view-controller (MVC) model [113] is one such object-based architectural model. It is implemented in the Smalltalk-80 [75] object-oriented programming environment as object classes. An interactive system is formed as a collection of objects structured as MVC triads. The model implements the functionality of the application, the view manages the graphical or textual output and the controller manages the user input. The model corresponds to the functional core and functional core adaptor of the Arch model, while the dialogue, presentation and interaction toolkit layers are encoded in a view-controller pair. A single model may have many view-controller pairs as ‘dependents’, as shown in figure 2.2. A single monolithic model can be associated with many dependent view-controller pairs [113]. The controller handles user input and modifies its model via a method call. The model may receive input from any other object in the system via a method call. When its value changes it notifies all its

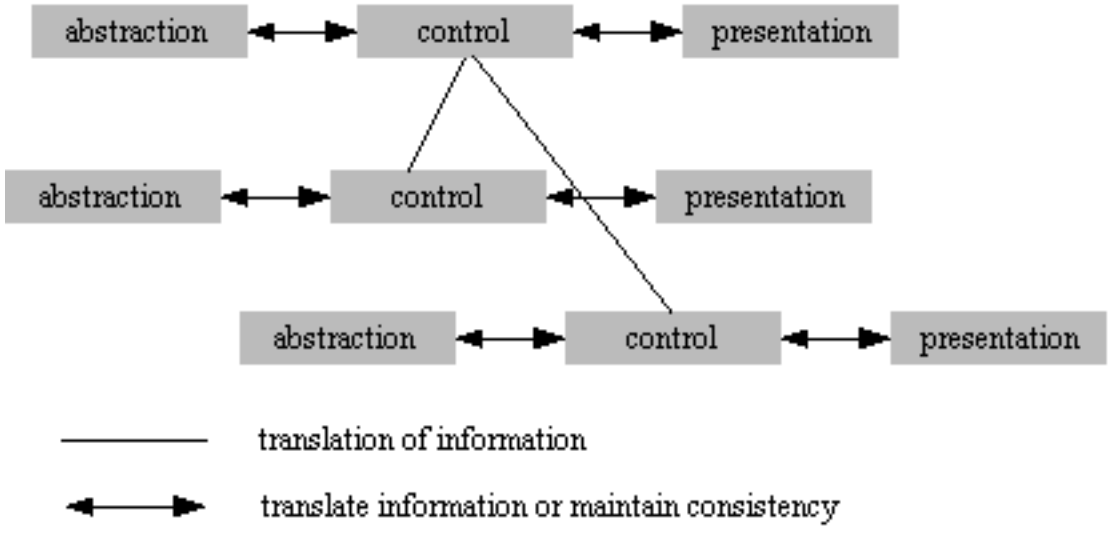


Figure 2.3. The PAC model structures the interface system as a composition of PAC agents, but does not define a particular communication and control mechanism.

dependants that it has changed, but it is up to them to update themselves accordingly. The view and the controller are coded with a specific model in mind. The model is coded without regard to its views and controllers. MVC supports the modular construction of user interfaces and a consistent scheme for the communication of information between its components. The application and the interface are collections of objects, and MVC does not prescribe a particular structure for these collections.

2.5.2 The PAC architecture

The presentation-abstraction-control (PAC) model [39, 15] structures a UIS recursively as a hierarchy of PAC triads, called *agents* in PAC parlance. The functional core adaptor is implemented by the abstraction component of the triad. The presentation component of the triad corresponds loosely to the presentation and interaction toolkit components of the Arch reference model. The dialogue control is made explicit in the controller component which manages the relationship between different PAC triads. The controller may itself be a PAC hierarchy. As a conceptual architecture PAC favours a modular decomposition of the user interface system and it provides more guidance than the MVC model about how to form the interactive system as a composition of PAC agents. Each PAC object is associated with its own abstraction which is only accessible to others through its controller. This favours an explicit representation of the communication between agents, contrary to the MVC model where different triads can communicate through a shared model. The PAC architecture is not supported by any UIMS. It does not provide a 'library' of basic re-usable components and it does not prescribe a particular control and data propagation mechanism. However, it is assumed that the controller components pass information up and down the composition hierarchy, each time performing some transformation of this data.

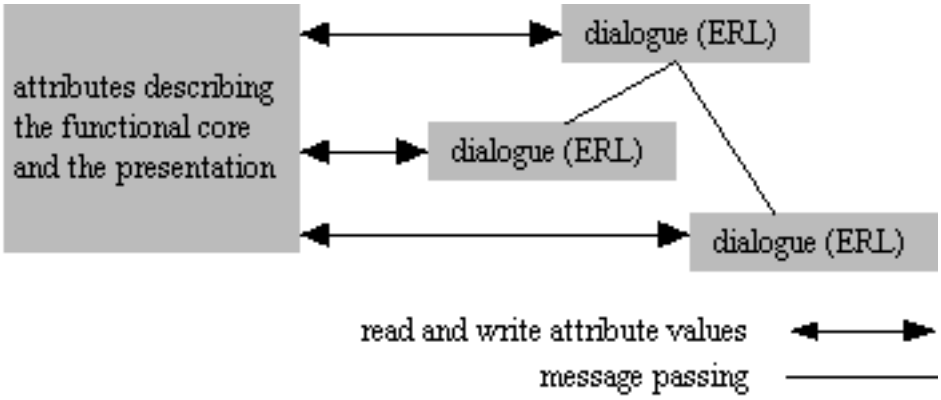


Figure 2.4. The TUBE composite object architecture. Tube does not support a modular decomposition for the abstraction component.

2.5.3 The composite object architecture

The composite object architecture supported by the Tube UIMS [93, 94] is an interesting example of a concrete architecture. It is similar to PAC in that the interface is built by composing objects into a tree-like hierarchy. Each object specifies some presentation and some dialogue control. An object is a lightweight process with some attributes defining its state. Its dialogue is described in the event-response language inherited from the Sassafras prototype UIMS [92]. Tube provides a basic set of primitives for building interactors and a library of commonly used interactors. It also includes a library of common attribute functions used to express constraints between object attributes. The dialogue is explicitly composed by passing messages up and down the composition hierarchy. Control is distributed among the interactive objects. The functional core adaptor is defined by a set of variables shared between the interface and the functional core. The communication control may thus be shared between the interface and the application. Relationships between objects or between the functional core adaptor and the presentation are encoded as constraints. Each interactive object may access the functional core independently from its position in the hierarchy via the attributes shared by the interface and the application.

2.5.4 The ALV architecture

The Abstraction-Link-View (ALV) architecture, of Hill et al. [95], is a follow-up to Tube, which is also very similar to the PAC architecture. For example, their abstraction components have the same designated role which corresponds to the functional core of the Arch reference model. The view component of ALV is similar to the presentation of PAC, and the link of ALV is similar to the controller of PAC. An important difference between them is that ALV is not simply a conceptual architecture, but has been implemented in the Rendezvous UIMS.

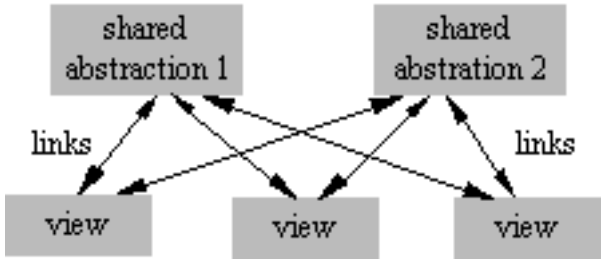


Figure 2.5. Abstraction-Link-View architecture (adapted from [95]). A view may be linked to more than one abstraction and vice versa.

ALV focuses on multi-user interfaces for which the separation of the functional core and the user interface system is very important. ALV supports the separation of the Abstraction and the Views, by storing in each View all the information it needs from the Abstraction. The Abstraction holds data that is shared between the multiple users of the system. This means that there is a significant redundancy amongst Views and between each View and the Abstraction. The common information is kept consistent by the link component which is a bundle of constraints, whose purpose is to connect each view with the abstraction. Multiple views may be linked to the same abstraction (figure 2.5) and multiple abstractions may be accessed by the same view.

In the Rendezvous UIMS an interactive system is constructed by the composition of objects. Distinct composition hierarchies (trees) are defined for the abstraction and the view objects. The two are not isomorphic. The view hierarchy is usually more complex than the abstraction hierarchy and it closely resembles the containment hierarchy of the displayed objects. Links maintain consistency between the values held in the objects and maintain constraints between the tree structures (figure 2.6). The links may also be organised in a tree structure themselves.

Rendezvous is an object oriented system. The interface developer composes existing classes, using the same language that is used to build interfaces. Programmers who use the Rendezvous system, see no difference between building user interfaces and building user interface components. Both are assembled from simpler interface components and

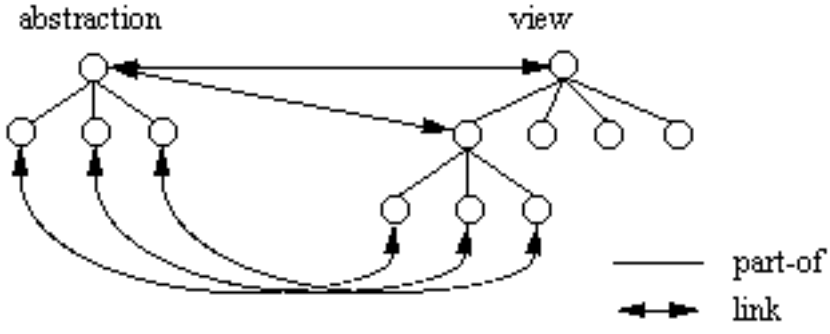


Figure 2.6. Composition of interactive objects in ALV. Abstraction and view objects are composed, and links are drawn across the two composition hierarchies.

the complete interface is a large complex interface component.

Rendezvous supports a declarative graphics system which handles screen updates and object selection and an event model which handles the queueing and the distribution of events. User input and reactions to user input are modelled by events. The communication between the abstraction and the view is specified as constraints between them and is supported by a constraint satisfaction system. All inter-object communication is supported by the constraints, and does not have to be described explicitly. This can be contrasted with the MVC architecture, where each View-Controller pair has to notify its model of changes and each model has to keep a list of its dependents. Thus, ALV is more modular than MVC.

2.5.5 The Garnet UIMS

Another UIMS that incorporates a constraint satisfaction mechanism is Garnet [136, 137]. Garnet emphasised the visual aspects of the interface for a single user environment. Similar to the Rendezvous environment, it contains an object system, a constraint system, a graphical system and a mechanism for input handling. Constraints may apply to attributes of objects that are displayed, or to objects belonging to the functional core. GARNET does not enforce an architectural separation between the functional core and the user interface system.

In Garnet, a few types of interactors are described through highly parameterised look-independent descriptions. Actual interaction objects are implemented through the instantiation of the parameters. However, it is hard to define new types of interactors [137]. All interactors have the same underlying state machine, and different dialogues are obtained by instantiating the state transitions. Myers et al. [137] suggest an analogy between the Garnet architecture and the MVC model. Application objects are analogous to the model, graphical objects correspond to the view and interactors correspond to the controller of MVC [113]. Garnet interactors differ from MVC objects or PAC agents, since they do not encapsulate their abstraction state or their displayed state. The interesting aspect of Garnet is how parameterisation is sufficiently powerful to express all graphical interaction. In subsequent chapters, when a formal architecture is discussed, a similar parameterisation will be adopted for certain behaviours. That parameterised description corresponds loosely to the state machine that underlies Garnet interactors.

2.6 Composition structures in object-based architectures

The object-based architectures share the view that the user interface system is constructed by a collection of objects. These objects, referred to as interactors below, consist in a structure of dedicated lower level components whose purpose corresponds closely to the components of the Arch reference model. The idea of a homogeneous object space [41], mentioned in section 2.4, is not always realised in these models as, in

most cases, the components of an interactor are factored into distinguishable collections that resemble a layered model. For example, in the ALV architecture, all abstractions are grouped together into a composition hierarchy and the same happens with the view components.

In some cases, e.g. in the Garnet UIMS, the scope of interactors is limited to the interaction toolkit and the presentation components of Arch. In most cases they model parts of the dialogue and the functional core adaptor components of the Arch model, as in ALV, PAC, and MVC. For MVC, PAC, and ALV, the functional core is distributed in the 'abstraction' or 'model' component of the interactors. Also, the presentation component of Arch is distributed in the presentation components for each interactor. Clearly, a single interactor may extend over all the components of the Arch model. Conversely, it is also obvious that the whole of an interface may be conceptualised as a single interactor, although it would be hard to implement it as one. Such a description would be very abstract and unstructured but may be refined by decomposition to a configuration of communicating interactors. Defining the composition of interactors that form the interface is the essence of the design activity. The architectures presented focused mostly on the opposite direction, i.e. the composition of interactors to form the interface architecture.

Common to all the object-based architectures is that the interactor may read and write two forms of data. One is particular to its abstraction and one is particular to its presentation. In one way or the other, the interactor maps changes of its presentation to some modifications of its abstraction, and vice versa. Finally, the interactor encodes some sequencing constraints on the order in which it will interact with its environment. These may be defined implicitly, as in MVC, or explicitly, as in PAC and TUBE.

The architectures discussed here differ in the way the objects are composed. In PAC and TUBE interactors are composed into a hierarchical structure. In PAC the controller component maintains the links to other 'nodes' in the hierarchy. In the composite object architecture of Tube, communication channels of the dialogue control components implement this hierarchical structure. In TUBE, the presentation and functional core adaptor components are described by a single set of attributes. The attribute set is structured in a composition hierarchy that reflects the geometrical containment of the objects on the screen. This hierarchy is mapped to the communication mechanism. In ALV the composition hierarchy for views is a containment hierarchy that is independent from the dialogue control. In MVC and Garnet the abstractions of individual interactors form part of a single monolithic description of the state, although this can be avoided in MVC. In PAC and ALV the descriptions of abstractions are kept distinct. In ALV the abstractions may themselves form their own composition hierarchy, and it is the job of the link component to maintain the consistency with the corresponding presentation composition hierarchy. PAC and Tube adopt a single hierarchy for the composition of interactors. ALV may support multiple composition hierarchies which may also be re-configured dynamically.

2.7 Conclusions

Software architectures for interactive systems take many forms. A starting point for most of the architectures discussed is to distinguish the functional core from the user interface system. Most architectures separate the two as far as possible, either in layers or as object-components in an object based model. The separation may be implementational or purely conceptual but in both cases it defines the scope of the interface design problem. Therefore, user interface software will be modelled as an entity that supports the communication between the user and the functional core.

Object-based architectures are the most promising approach in modelling interactive systems. Object-based architectures are not monolithic and sequential. They model the interface software as a composition of co-operating objects. These models are highly modular and support concurrency and distribution. This has several advantages concerning iterative design, support for distributed applications and support for multi-thread dialogues. As will be seen in the following chapters, this affects the choice of the abstractions used for the formal specification of user interfaces, which are also required to be modular.

The conceptual distinctions described by the Arch reference model for interactive system architectures, filter through to the components of object based architectures. The various models discussed define the roles of their components differently and differ also in the way these are put together to compose the interface. Clearly, the ability to compose user interface representations by composing component specifications is an essential requirement for the formal modelling scheme.

All the models discussed in this chapter were introduced by their authors informally or through a UIMS architecture. In subsequent parts of this thesis, the formal specification of user interface software is discussed. It will not be attempted to model formally any particular architecture of those discussed in this chapter. A formal model should be more general and abstract than any concrete architecture (e.g. ALV). By definition, the formal model is more precise than a conceptual model (e.g. PAC) and so it is debatable whether a formal interpretation of a conceptual architecture describes the same concept. It is clear though that the architectures discussed in this chapter draw some lines along which the formal interactor model will be defined. The examination of software architectures in this chapter suggests that the formal specification also needs to be object based if it will be at all relevant to the software it models. Similarities are also sought in the internal structure of the interactor objects and in the way they are ‘glued’ together to construct the composite formal specification of the user interface.

Chapter 3

Interactors: the concept and its evolution

This chapter reviews research into the application of formal methods to the study of Human-Computer Interaction. The review provides the context for introducing the interactor model of chapter 4, and records some lessons learnt from previous research approaches which have been instrumental in its development. The case is made for formal architectural models of user interface software, collectively referred to as interactor models. Current research in this area is reviewed and comparisons are drawn across the different models. The discussion identifies those elements which make an interactor model a valid abstraction of interface software and a useful engineering tool for the specification of user interfaces. Finally, the formal framework in which interaction is modelled in the remainder of this thesis is described and the LOTOS formal specification language is introduced briefly.

3.1 Specifications as a tool in the design of interactive systems

As mentioned in chapter 1, individual formal methods may differ in their syntax, the entities they model, the underlying inference systems, their intended role within the software development process, etc. [38, 190]. However, they all share a commitment to abstraction. Abstraction, it is suggested, should give rise to generic device independent components described with minimal detail [172]. Traditionally, the envisaged role of a formal method is to describe the essence of a system function, without premature commitment to implementation considerations [84]. By the use of sound engineering principles and rigorous reasoning, usable and effective systems may be designed. Formal software development is seen as an almost mechanical process of refinement or transformation. This view of software development has been repeatedly challenged within the human-computer interaction research community, for the following reasons:

- A statement of principles, entities or relations is a product of design activity just as much as the user interface software itself. The abstract formal specification may

equally well suffer from errors, prejudice, misinterpretations or influences from the notation they are expressed in [172].

- The details omitted by the abstraction activity may be crucial for the success of the user interface. Carrol [34] has argued that a representation of a design needs to incorporate all aspects of a design and in 'infinite detail'. The interface may itself be the only valid and usable codification of its design.

Took [172] suggests that design should be specification-centred as opposed to specification-driven. Duke and Harrison [57] characterise the role of the specification as integrative: it brings together contributions originating from different perspectives and allows the developer to check their consistency and possibly to identify issues which require further consideration. However, they also point out some limitations. Usability claims cannot be made within a formal system alone; the use of formal properties in rigorous software development does not guarantee that the result will be a usable system. Claims about usability must be validated by means other than formal verification. Further, they experienced difficulty in trying to communicate formal specifications to designers not accustomed to formal methods.

The arguments above show that a formal method is not a panacea for the engineering of interfaces. Considering the current state of the art in the area, it seems more appropriate that rather than trying to shape the design process, research needs to investigate where and how formal specifications of user interfaces can be shown to provide tangible benefits [87]. This argument is consistent with currently accepted views as to the role of formal methods in software development in general. Holloway and Butler [97] suggest that formal methods researchers should not attempt to dictate new methods and processes for software development but should attempt to find how their methods and tools may help within currently established practice. It is becoming increasingly accepted that formal methods are a supplementary and not a replacement technology for software development [26].

The argument above suggests that formal methods can be useful for user interface design but with the following qualifiers:

- The design of an interactive system should not be driven by the specification but may use it as a means of representing designs and assessing decisions.

- The choice of the formal specification can enable the designers to articulate their decisions at the 'right' abstraction level. A formal specification notation is good for a specific job and for a well defined scope of the problem domain. Not all design decisions but, in fact, just a few may be based on the formal specification. Accordingly, the scope for the representation techniques investigated in this thesis is only the user interface system as defined in chapter 2.

- Formality requires tool support, for example to ensure syntactic correctness and semantic consistency. The validation of the specification may be aided by assessing

the plausibility of the inferences drawn from the specification or by inspection of a simulation for executable specifications.

A specification may address various levels of abstraction and a high-level prototyping language may look no less abstract. What distinguishes the formal specification is the mathematical definition of its semantics and the underlying inference system associated with them. Ideally, the specification should make it possible to verify or to assert properties of the system which it is not possible, or more costly, to observe in an implementation. In doing so, it may aid the designer to investigate or to prescribe qualities of a design. It is not always necessary that an implementation should be constructed by the refinement of the specification.

In the thesis this limited use of formal specifications is adopted. The thesis investigates them as potentially useful tools for the design of user interfaces. It does not propose a model for the design activity. Also following the above arguments, it does not propose to localise design decisions at a particular level of abstraction, recognising that design takes place at many abstraction levels and throughout the development.

3.2 Structure and abstraction level in the specification of interactive systems

For the sake of presenting and discussing the various approaches to the specification of interactive systems, two dimensions that characterise them are introduced. These are the level of abstraction adopted and the degree of structure used in the specification.

A high abstraction level refers to general properties of interactive systems with as little commitment as possible to a specific representation or problem. The lowest level of abstraction corresponds to a fully functional implementation of a particular system.

The degree of structure adopted is a reflection of the specificity of a model. A general purpose specification or programming language has a low level of structure. Some structure is introduced by adopting personalised techniques or stylistic conventions. In an attempt to standardise these techniques, specification styles have been introduced [180], (they are discussed more extensively later in this chapter). The consistent use of these specification styles helps teams of specifiers work together and provides a conceptual aid for the specifier. At the other extreme, an object-oriented system [131] has a very high degree of structure, capturing generalisations of objects into classes and similarities between classes into an inheritance hierarchy.

The classification along these two dimensions, illustrated in figure 3.1, helps present some of the most influential works in this research area. Structure and abstraction level correspond to the two axes of the Cartesian plane. Different schemes for the formal representation of user interface software correspond to areas on the plane, indicated accordingly. There is no strict boundary between the areas indicated, as there can be many similarities between neighbouring approaches. The mapping to the Cartesian

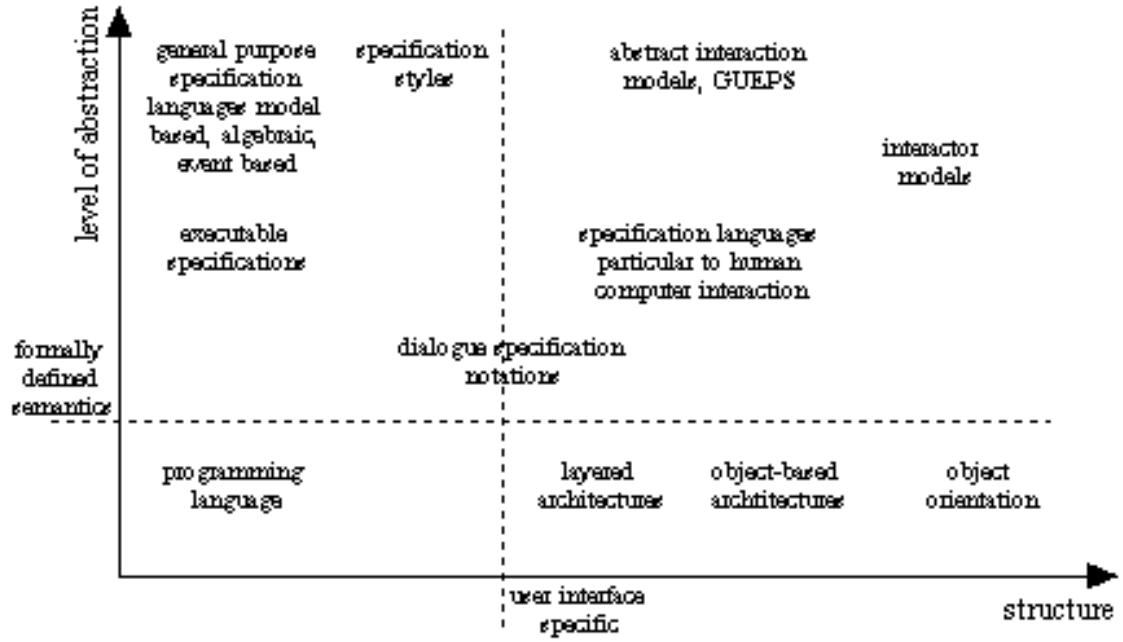


Figure 3.1. Mapping formal specification techniques to dimensions of abstraction and structure

plane is suggestive of the role of interactor models and their relation to other approaches. Arguably, structure and abstraction levels are not orthogonal dimensions, since the introduction of structure can represent some implementation bias. To an extent this is correct, but as the exposition below shows it is possible to provide highly abstract structured descriptions which provide no hint as to the way the software will be implemented, e.g. the Agent and Interactive Processes models discussed later in this chapter.

Low levels of abstraction correspond to concrete implementation languages and implementation architectures. Layered and object-based software architectures are characterised by a high structure and, where object orientation is supported in such a model, the structure is maximised. As more structure is introduced in a representation, whether that be formal or not, the description moves closer to an architecture or an implementation.

Interactors are more abstract than the object-based architectures of chapter 2. Both are highly structured. The similarity in structure is not coincidental. Historically, both interactor models and object-based implementation architectures for interactive systems were proposed to model modern graphical and multi-modal interfaces for which the less structured models had been found to be unwieldy. Figure 3.1 probably overplays the role of interactor models, assigning a large area to them, simply to make the point that they may be used with varying degrees of structure and at different levels of abstraction. As the detailed exposition of the following sections will show, it is not easy, nor is it accurate, to draw a line that distinguishes interactor models from abstract interaction

models. In part, interactors have developed out of research in abstract interaction models.

3.3 Dialogue specification notations

Traditionally UIMS research has focused on the description of the dialogue component of the user interface. This emphasis was a consequence of the nature of the early user interface systems which supported mostly command language interaction. The emergence of object-based architectural models for user interface systems has blurred the distinction of what is and what is not dialogue. However, as the brief overview of some of these systems has shown, many UIMSs propose a particular formalism for the description of the dialogue. In the context of an implementation architecture the dialogue specification notations are special purpose programming languages. This section examines those which have also been associated with a formally defined semantics to serve an analytical use.

An early comparison of three dialogue models is presented in Green [78]. Green compares the expressive power of state transition networks, context free grammars, and a simple event based model. The event based model that he uses adopts more programming language constructs than the other two notations. It is shown to be more expressive than the other two, which are found to be more or less equivalent to each other. Comparisons between the expressive power of various specification notations are informative results and for most specification languages there will be some information already available in the literature, e.g. [82]. However, such comparisons are possibly misleading. Green points out, that the effectiveness and the ease with which the notations are used are perhaps more appropriate measures for their assessment. These attributes are hard to gauge, as they depend on the context of application, the purpose of using the notations and the persons who use them.

State transition networks (STN) were one of the first notations to be used for the specification of interactive dialogues, e.g. [182]. They are relatively simple and widely understood by computer scientists. STNs come with several variations of expressive power, e.g. recursive transition networks, or augmented transition networks (ATN), e.g. [103]. An STN is a simple network with labelled nodes and arcs indicating states and transitions between them, while an ATN may include indications of the display, a description of the state by a set of variables, conditional transitions, etc. Jacob [103] uses small-size ATNs to specify individual interactive objects. Their composition is not formally modelled, but implemented as a co-routine structure, whereby individual ATNs communicate via procedure calls.

The main limitation attributed to STNs is that the expression of multiple threads of dialogue may lead to a combinatorial explosion of the number of states. For this reason, other diagrammatic notations more appropriate for handling concurrency have been proposed. The most prominent are Statecharts [81, 183] which introduce the notion of

'parallel' states and Petri-Nets which are particularly appropriate for representing concurrency [17, 144].

Grammars were originally very popular for specifying text based interaction. They have similar expressive power to STNs and are equally well documented and understood by programmers. Grammars focus on the representation of user actions and they provide very concise descriptions of input sequences. They are not as good at representing concurrency and do not support an explicit representation of state. Studies like that of Green [78] have pointed out the weaknesses of using grammars for modelling graphical interaction, so their use tends to be rather scarce recently. An interesting approach reported recently, called DIGIS [42, 43], combines regular expressions, which are a simple form of grammar, with an event based process algebra. The result is a hybrid notation which capitalises on the familiarity of programmers with grammars and has sufficient expressive power to specify and program graphical interaction.

Production systems are a textual event-based notation which has been used successfully in modelling direct manipulation dialogues [142]. The specification is made up of a list of rules which specify a condition under which they fire and actions to be taken when they do. The actions may refer to user activity, internal communication among interface components or output events addressed to the user. Production systems are good for modelling interleaved but not sequential activities. The Tube and Rendezvous UIMSs, discussed in chapter 2, use the Event-Response Language which is a special case of a production system. Production systems have had limited use as a specification notation [143]. Recently, Abowd et al. [3] have proposed their use as a front-end notation to a state transition network, which would allow the expression and verification of usability related properties.

This review of dialogue notations is not detailed and does not favour a particular formalism. Dialogue specification notations are well understood and the debate which surrounded them in the early days of UIMS research has concretised in some understanding as to the trade-offs made with each. In general, event-based models are more appropriate both at the specification and the implementation level. As this brief overview has shown, event models may be accommodated in a variety of formalisms, visual and textual. The target area of interest, personal taste and existing tool support may justify a particular problem-specific choice. More relevant to this thesis is the choice of an event model, whether it supports interleaving or true concurrency, whether it supports a model of time, etc. These options are discussed more extensively in the remainder of this chapter.

3.4 Using general purpose specification notations to specify interactive systems.

Early approaches to the specification of user interfaces used general purpose specification languages. For example, Sufrin [168] used Z for the specification of a text editor and a similar specification was written by Ehrig and Mahr in an algebraic

framework using ACT-ONE [61]. Chi [35] reports the use of four different algebraic notations for the specification of a simple calculator. Several approaches using logic for the formal specification of user interfaces have been reported, e.g. [91, 105, and 159].

The contribution of such work is mainly as model-specifications for researchers and practitioners. The specifications provide examples as to the choice of appropriate abstractions but the lessons learnt are more or less confined to the scope of the example, e.g. text editing. Such case studies aimed to test the hypothesis that formal methods of specification may be usefully applied to interface development. Because of their limited scope and academic nature, they do not on their own validate this hypothesis which still remains an open research question. These case studies also aimed to assess the appropriateness of a particular specification notation for the specification of interactive systems. For example, the experiment of Chi included the comparison of algebraic notations with Z to specify a pocket calculator interface. It revealed how it can be cumbersome to use formal notations to specify interfaces but also argued that there are some gains, for example, speedier implementation after the specification, early evaluation and insight into design drawbacks, etc.

There are a number of problems in using notations such as Z or VDM to describe user interface designs. They do not lend themselves easily to specifying the flow of control in a user interface, or the syntactic regularities which characterise the observable to the user behaviour of a system. The dialogue notations, discussed in the previous paragraph, are particularly appropriate for this purpose. As was pointed out, the specification of human computer dialogue is better served by event-based notations. For this reason event based formalisms have been increasingly adopted for the specification of user interfaces, e.g. CSP [96], Statecharts [81], etc. Unfortunately, while these notations are useful for dialogue design, the formal specification of user interfaces often requires the description of functionality for which they are not particularly well suited.

This argument has led to the suggestion that interactive systems need to be described by hybrid specification languages that combine an event-based element for the description of behaviour and an algebraic or model-based component for the description of functionality. Marshall [129] describes a hybrid technique which uses a diagrammatic notation based on Harel's Statecharts [81] to describe the flow of control at the user-interface, and VDM to specify the operations of the system. The formalism was not shown to cope well with parallelism and the resulting specifications were not well structured, in that the state specification was monolithic and made reading the specified sequences of interactions very hard [1].

A hybrid specification language called SPI (Specifying and Prototyping Interaction) which is particularly relevant to this thesis is summarised by Alexander in [8]. CSP is used to describe the dialogue structure for the interface in terms of the sequencing, synchronisation or interleaving of event occurrences. The CSP elements are extended with a specification of the semantics of each event which simulates the event occurrence in simple prototyping environment. To specify the semantics of each event Alexander has experimented with two languages, the C programming language and the me-too

functional programming language. A consistent specification technique is supported which consists in representing each interaction object on the screen as a CSP process. The display is specified (statically) as the parallel composition of the processes corresponding to the objects it contains. SPI demonstrated the potential of CSP for the formal specification and analysis of dialogues. However, the approach is semi-formal as there is no mechanism to ensure that the event semantics, specified in the C language or me-too language, are consistent with the CSP process specification. Clearly the use of a hybrid specification language like LOTOS could overcome this problem although with the loss of the prototyping facility. From a formal specification point of view, SPI demonstrates a style in structuring the specifications which is used also in the specification of logical input devices and of interactors discussed in later sections of this chapter.

As a prototyping language SPI is very similar to Squeak, by Cardelli and Pike [33]. Squeak is a programming language specially designed for programming user interface components. It combines constructs of CSP and the C programming language. It demonstrates how logical concurrency facilitates the programming of user interfaces.

The work of Marshall and Alexander indicate that interface specification is better served by hybrid notations. Since then considerable advances have been made with the development of hybrid specification languages and the development of tool support. Hybrid specification languages were developed to address the perceived need for a language that captures both the event based nature of the behaviour and the semantics of the operations triggered by these events. There is much to be gained if existing and widely disseminated general-purpose notations are used. In particular, gains can be expected from well developed theoretical results, published case studies, a wide community of users, and tool support.

In recognition of the value of hybrid specification languages, LOTOS [18, 100] is adopted in this thesis. This follows an early experimentation with CSP [96], reported in [121]. LOTOS is an international standard [100], with rich tool support and a wide community of users. The required specificity for the domain is attained by defining structures appropriate for the specification of user interfaces, rather than defining a new specification language or extending the semantics of the LOTOS language itself. To help in reading the thesis, LOTOS is introduced briefly in section 3.8.

The next two sections overview research that has attempted to define appropriate generic abstractions for the description of user interfaces. This research has not been concerned so much with developing notations, although the necessity for hybrid specification notations persists when the abstractions are proposed as tools for interface design representation. The abstractions discussed have become increasingly concrete over time, and throughout their history they have been expressed in a variety of formal frameworks.

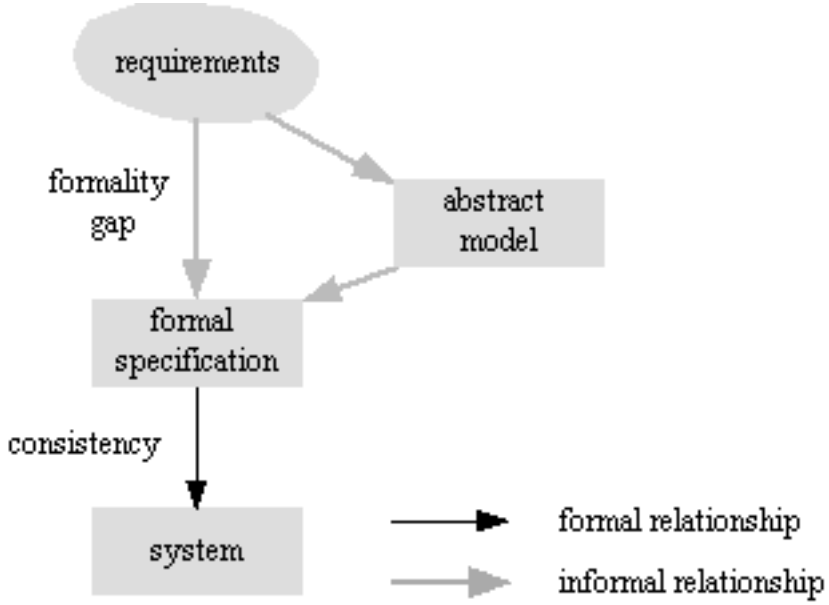


Figure 3.2. Abstract interaction models can help bridge the 'formality gap' between informal requirements and design specifications (adapted from [49])

3.5 Abstract Models

Abstract models of interactive systems emerged from research into the concept of *generative user engineering principles* (GUEPS) proposed by Thimbleby [170]. These principles were called generic to indicate their applicability in a wide range of systems. GUEPS were to be given both a formal and an informal description and they were intended to be used generatively, i.e. as design constraints. Examples of such principles are that it should be possible to predict the effects of commands, that the display should let the user observe the internal state of the system, etc. Bornat and Thimbleby [21] report an informal application of such principles in the design of a display editor called *ded*. *Abstract system models of interactive systems*, (for short abstract models), were developed in an effort to formalise GUEPS in a manner general to interaction and not to a specific interactive system [86]. Dix [49] suggests that abstract models could help bridge the gap between the informal domain of user requirements and the formal domain in which some requirements have been formally specified (see figure 3.2).

Abstract models are essentially meta-models, i.e. higher level abstractions, which can be specified formally in whatever specification notation is used within a design project. Dix and Harrison [49, 84] use general set theory for their description rather than any particular specification notation. When principles are expressed using the abstract models, any commitment to a particular implementation technique or system architecture is avoided [84].

3.5.1 The state-display model

A family of models falls under this title. They describe interactive behaviour in terms of the relation between the internal workings of a system and how these are reflected on the display. These two distinct views of a system are modelled as two separate systems or layers. Usability related properties can be conceptualised and formalised as properties pertaining to the relationship of these two layers. This two-layer approach can be extended to multiple layers, e.g. physical system, digital control system, system metaphor [83].

The internal system layer is modelled by a set of states S, an initial state s_0, and a set of commands which transform states C $S \rightarrow S$. The commands refer to the inner functionality of the system, without concern about how this functionality is invoked by the user, or what visible effects it gives rise to. For example, a text editor has commands to edit a document, to insert characters, to move a cursor, etc. The display layer is described in a similar fashion to the internal state layer, by a set of display states D and the operations on displays O $D \rightarrow D$. The display layer may have commands to move the cursor, scroll a window, insert characters, etc., although these commands pertain to the displayed representation of the text rather than its internal representation.

An example of a usability related property which can be formalised with this model is the *visibility* of the system. It concerns the ability to associate what is displayed with what is actually happening in the inner system. Visibility is formalised as the conformance between the state and the display layers [86]. The states of the two layers are related by relation *render* r $S \times D$. The operations on states and displays are related by relation *compatible* co $C \times O$ which defines pairs of compatible operations for the two layers. The two layers are *conformant* if for any <c,o> co and <s,d> r it is the case that <c(s),o(d)> r. For any pair of elements which are related by r the images of the two, obtained by compatible operations, are also related by r. In other words, it requires that the state and the display are mirror images of each other and that the compatible operations upon them, with respect to relation co, have compatible effects, with respect to relation r.

This state-display conformance property is too strong to be a design constraint, as it is not usually possible for the display to reflect all the information in the state. Several refinements to this simple state-display model have been proposed, which aim to make it weaker and therefore more practical. One solution investigated in [160, 161] was to use some sort of filter, to focus on those components of the state and the display which are relevant for a particular task. A task t can be associated with a projection on the state $attr_s(t):S \rightarrow S_t$ and the display $attr_d(t):D \rightarrow D_t$ Visibility is defined relative to a task t using relation v(t) $(attr_s(t)S \times attr_d(t)D)$ and relation co(t) $(C(t) \times O(t))$. A formal model for the definition of these projection functions, called *templates*, was defined in Roast et al. [160]. Templates are an attempt to bridge the gap between formal interaction modelling and the results of a task analysis. This research, reported extensively in [159], aimed at providing a framework for the integration of psychological analysis and formal interaction modelling.

Another way to weaken the requirement of state-display conformance is to think of the display as an approximation of the state, defined as a mapping $v:S \rightarrow D$. This approximation preserves a partial order between states, i.e. the view mapping should be monotone with respect to partial orders defined over states and over displays. Rather than relating the instantaneous display to the state, Harrison and Dix [84] suggested that the state can be uncovered as a series of distinct views which combined reveal the underlying state. The relationship between state and display should be expressed in terms of a mapping $v^+: S \rightarrow D$ which they call *panoramic view* (or the *observable effect* in the context of the PIE model [49]). Harrison and Dix [84] have proposed a construction of the mapping v^+ in terms of the repeated application of the simple view mapping on successive states. They define a system to be *observable* when the entire state is visible through the panoramic view and when the panoramic view contains an unambiguous representation of the component of the state which is relevant to the task of the end-user. A command is called *visible* if it modifies visible data only and produces results which are visible through the panoramic view.

Another abstract principle which can be expressed in terms of the simple state-display model is the *predictability* of a system. A system is predictable if it is possible to tell what available commands will do on the basis of what is currently perceivable [86]. A system is called predictable if for a pair of state (s,s') S such that r(s)=r(s') it is the case that f(c(s)) =f(c(s')) for any c C, where f is a function that extracts some features from the state. If f is the identity function then a strong definition of predictability is obtained where a view determines the effect of commands on the state. Different choices for function f may give rise to various expressions of predictability. Variants of this concept have been proposed for the more structure system models examined later in this chapter.

Abstract models help formalise essential intuitions regarding interactive systems. For example, the notion of a panoramic view describes an essential property of interactive systems, i.e. that while not the whole state is displayed the user can reveal it by interaction. A close examination of the formalisations above shows that several assumptions are made regarding the user. The definition of visibility is founded on the idea that the user constructs mentally the panoramic view from the individual views displayed. The definition of predictability may also be taken to assume that the user will notice infinitesimal changes to the display. Similar assumptions are associated with most formalisations of interaction properties found in the literature. The first opportunity is taken here to point out that caution should be exercised as to the assumed psychological validity of the expressions and to the feasibility of their realisation as properties of an implementation.

The next question concerning the thesis is the validity of the state-display model as an abstraction of modern user interface systems. Recall that the inner functionality of the system was described by a set of commands. In [84] the inner system is assumed to receive these stimuli through, e.g. some function ‘parse’ which interprets user actions as commands independently of the display state and a function ‘run’ which collapses sequences of commands to single state transitions. This structure does not reflect accurately the nature of direct manipulation and this is revealed in the formalisation of

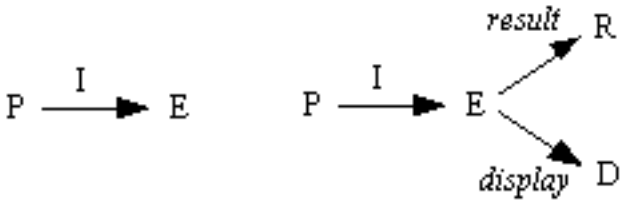


Figure 3.3. Illustrations of the PIE and the red-PIE models.

temporal property, Harrison and Dix [84]. This direct manipulation interfaces, requires the temporal ordering of physical inputs to reflect the ordering of command invocations. Harrison and Dix point out the example of character insertion and deletion in a line editor: there is no mapping from input sequences to commands as the effect of the input is dependent on the display state. This dependency is not captured by the state-display model, which does not portray how the interpretation of graphical input depends on the contents of the display. As Took [174] has argued, the formal model should capture the *syntactic* dependency of input and output. It is hard to model interaction when commands are interpreted on the basis of the input history only.

directness by attributed to

The state-display model is a framework for the definition and articulation of properties of the use of interactive systems. It is not intended to be used constructively in the development of interactive systems and it is difficult to do so because of its lack of structure. A constructive application of the state-display model has been reported [106], where a temporal logic is used to specify safety and liveness requirements at a very high level of abstraction. This specification is refined to an executable prototype by specifying the behaviour of a graphical interface. The designer is offered a logic programming language to specify interaction but no structure or guidance as to how best to do so. The limitations of the state-display model as a design tool have prompted research into more constructive models, namely the Agent model of Abowd [1] and the interactors of Duke and Harrison [55] discussed in later sections of this chapter.

3.5.2 The PIE model

The PIE model adopts a ‘surface philosophy’ for the description of the interactive system (see [49] chapter 1), i.e. it does not delve into the internal workings of a system. An interactive system is seen as a ‘black box’ to which input is given and of which output may be observed. The model relates user *programs* P, which are sequences of commands, to the *effects* E they might have, via an *interpretation* function $I:P \rightarrow E$, (see figure 3.3). The effects may refer to the display, the entire information managed by the interactive system, etc. Properties of interaction can be expressed abstractly in terms of constraints on these components, without referring to any internal representation of the system. An important result that comes out of this attempt at a behavioural specification of a user interface, in terms of the function I, is that it is always possible to describe the PIE model in terms of a state-based description which is no less abstract than the functional description. Therefore, state based and behavioural representations can be used interchangeably according to the context (see [49, chapter 2]).

The PIE model affords concise definitions of concepts such as the predictability of the system, reachability properties, undoing commands, etc. The PIE model has been extended to the red-PIE model to discuss the relationship between the display (that which the user sees) and the result (that which the user wants to achieve through interacting with the system). The red-PIE model includes the definition of two mappings from the effect space to the result and the display (see figure 3.3). These may be interpreted respectively as the portion of the system state which is interesting for the user and the part of it which is displayed. The properties that are described in terms of the red-PIE model concern mostly the observability of an interface, i.e. what can be inferred about the result from the display.

Observability properties pertain to whether information is present on the interface or possible to reveal. They do not relate to the user's competence, i.e. how easy it is for the user to examine this information and they do not give rise to performance predictions. Similar limitations apply to the predictability expressions. Dix [49] proposes three approaches to overcome these limitations of abstract principles:

- Informal reasoning can be used that can draw from other disciplines, e.g. psychological analysis.
- Richer models may be produced. The increased structure in the model captures more of the notion of ease of use. Dix himself takes this approach, and this has led to more detailed and refined expressions of the principles, but has not improved their validity, [49].
- Richer models may be produced which incorporate some aspects of user modelling. Examples are the approach of Roast [159] and at a more concrete level the '*syndetic'* modelling of Faconti and Duke [63], discussed later in this chapter. The hardest problem in these cases is that some aspects of the user model have to be formalised.

Further to the description of abstract principles, the PIE model provides a general framework for the discussion of a range of issues like temporal behaviour, non determinism, interference between different windows on multi-window systems, etc. The most definitive and comprehensive treatment of the PIE model can be found in [49], which incorporates and integrates earlier publications on the subject.

By construction, PIE models the interaction of a single user and a single machine. It does not cope well with multiple input streams which may be the case with contemporary multi-modal systems. [49] discusses several ways for modelling direct manipulation systems. Interaction is seen as display-mediated, i.e. the system is described in two layers with one PIE describing the display and another PIE describing the inner system.

The PIE model has exercised significant influence on later formal models of interactive systems. Because it is so abstract, it is not sufficient to be used as a basis for the design of interactive systems. However, the models that follow it can be thought of as instantiations of the PIE model in a particular formal framework, that each emphasise

some aspects of interaction. Looking back at figure 3.1, it is now easy to explain why it is difficult to draw a line between abstract models and architectural models. The PIE model serves its aim to provide a framework for describing interaction properties. However, as pointed out in [1, 54], it does not model explicitly the meaning of individual operations, it gives an unstructured description of the interface. The need arises for a constructive notation that will allow the description of the user interface as a composition of formally specified autonomous units.

Using the PIE model amounts to extending and specialising it on a problem by problem basis. Modifications of the model may sacrifice its simplicity for extra expressive power. One possible constructive use of the model is in terms of a layered design of an interactive system, where the 'outer' layers represent the physical interaction and the 'inner' levels represent the functional core.

Runciman [165] proposed a functional implementation of the PIE model and defined program transformations that help develop prototypes of the interactive system. However, the functional specification of an interactive system can be quite cumbersome. Runciman already notes how the validity of the specifications depends crucially on the evaluation strategy used for their interpretation ('lazy evaluation' in [165]). Took [174] pursues this point further. He argues that functional specifications are inherently inappropriate for specifying graphical interaction. One of their fundamental characteristics is what is called the Church-Rosser property: the interpretation of a functional program may proceed in any order. Any interleaving of input and output is semantically acceptable for a functional specification of an interactive system. Functional specifications do not support any notion of intermediate results unless a restriction is put on the evaluation order. This is not a problem in representing and manipulating mathematical properties of the end-result of some computation, but is totally unacceptable for specifying an interactive system where the user is interested in a possibly open-ended sequence of intermediate results. The same reason underlies the inability to formally specify in a functional language that a particular output may only occur after some input, i.e. what could be called an output trigger operation.

Took [174] raises one more objection to functional specification of the PIE model. PIE models the user input as an argument and the output as the result of an interpretation function. The semantics of this function encode the dependency of input on output. Dix in [49, chapter 6] discusses the specification of direct manipulation systems and points out how it is difficult to build into the interpretation function the 'knowledge' of the display at any moment in interaction. It is easier to encode the dependency of input on output syntactically, so Took proposes an algebraic formalisation of PIE which models input and output as operands of an interpretation operation. A similar approach is taken in the next chapter. The relationship of input and output and the interpretation of input are modelled algebraically. Contrary to Took, the synchronisation of input and output is modelled by a process algebraic specification.

3.5.3 Interactive Processes

The interactive processes model, proposed by Sufrin and He in [169], is an interpretation of the red-PIE abstract model in the framework of state-based processes. Crudely, these may be described as processes whose events correspond to transitions within a state-space. The state-space and the transitions are specified using a model-oriented specification technique, Z in this case. An event occurrence is constrained by the conditions associated with a transition, but also, by predicates upon the traces of events of the process. An Interactive Process is driven by the user who is assumed to be ready to issue a command at any instant. On the contrary the system will only participate in a command when it is ready for it. An Interactive Process is also associated with view and result mappings similar to those of the red-PIE model. Subsets of the 'show' commands have the effect of displaying the view to the user or yielding a result.

Some properties of interactive systems, originally introduced as expressions of GUEPS in the context of the abstract models, have been formalised in the framework of Interactive Processes [169]. They are described in terms of relationships between histories of commands. Two command sequences are called *result equivalent* if they cause the same set of results. They are *view equivalent* if they cause the same set of views. If two equivalent command sequences can be extended indefinitely without becoming inequivalent then they are called *indistinguishable*. An example of an observability property expressed in these terms is *visual consistency*: a system is (strongly) visually consistent if its view indistinguishable command sequences are result indistinguishable. Sufrin and He [169] propose this last expression as a formalisation of the slogan 'what you see is what you get'. They proposed a host of similar expressions to describe GUEPS related properties, like predictability, honesty, trustworthiness, etc.

The Interactive Processes model was documented in [169] by a set of small examples, in which all the commands determine the result and have an immediately visible effect, i.e. they all belong to the set of 'show' commands. They referred to command line interfaces and, as with the PIE model, there are reservations as to how this model could be applied to direct manipulation interfaces. In this kind of interface, the interpretation of user commands depends on the display context. In the Interactive Processes specification there is no separate representation for the display context. If the display affects the interpretation of user commands, the dependence is 'distributed' in the semantics for the state-transitions. As mentioned already, in direct manipulation the dependence is syntactic [174] and it is easier and more economical to model it as such. The Interactive Processes model has a powerful mechanism for the specification of the dynamic behaviour of the process. This combines predicates upon traces with the state-based constraints on individual event occurrences. Complex behaviours can be defined constructively by means of a set of operators that construct sets of traces as in the CSP specification language [96].

The Interactive Processes model is useful as a formal framework for expressing abstract design decisions, but it is not so practical for the specification of interfaces. The monolithic description of the state is unstructured and may contain irrelevant detail, as

far as a particular task is concerned, and this may make it quite cumbersome as a design notation. Sufrin and He [169] point out that the proof of some of the interaction properties can be very complex for non-trivial systems.

3.5.4 Agents

Abowd [1] proposed a development of the Interactive Processes of Sufrin and He to a constructive model which aspired to be a usable design tool. Abowd adopted a structured description of the state as a mapping of attributes to values. Similar to the red-PIE model and the Interactive Processes model, Agents distinguish between the result and the display. While Interactive Processes describe the output as a stream of display events Agents capture the persistent nature of the display in terms of attributes and their values.

An Agent specification consists of three components. The internal specification describes the state and the operations on the state. The communication component maps events which constitute the externally observable behaviour of the Agent to the operations upon its internal state. The external component specifies the set of traces of events which are legal behaviours for the Agent. The roles of the internal and the external components overlap and this hampers the effectiveness of the model. A particular event sequence which may be specified in the external component, i.e. it is one of the possible traces of events specified, may be ruled out by the pre- and post-conditions related to the operations which are effected by these events on the internal component. This makes it a likely possibility that unsatisfiable specifications are written inadvertently. The flexibility of the Agent language, which allows the partial specification of event ordering in any of the two components, may be counter-productive.

Abowd defines two types of composition operators. The interleaved composition of Agents joins their state spaces, while a synchronised composition models the communication between two Agents. The purpose of these operators is to help build complex specifications from lower level ones, although their use seems to be quite involved. Abowd defined a design notation by which the Agent is specified by filling in various slots which describe its attributes, their types, their initial values, as well as a constructive definition of its traces using a subset of the CSP process algebra.

Abowd proposed expressions of GUEPS related properties following the approach of Sufrin and He [169] discussed above. Similar to the model of Sufrin and He, it is difficult to apply the Agent model in practice for anything but trivial examples and the contribution is mostly theoretical. On a methodological note it is not clear how to decide what should be modelled as an Agent. In the example applications of the Agent model, operations are bundled together not because of their correspondence to a particular interactive object, but rather to reflect similarities in their definition. It seems difficult to generalise from the examples of [1] to a general understanding or method for structuring the interface specification.

The Agent model is an attempt to develop a compositional abstraction for interactive objects, which is oriented towards a constructive use. It is a move towards architecture oriented models, as it incorporates some structure to the state and display representations. Properties of interactive systems which were defined abstractly with the red-PIE model and the Interactive Processes model can be related to more concrete system specifications. Like the SPI specification environment of Alexander [8], Agents advocate the usefulness of hybrid notations whose components can address the diverse requirements of specifying interactive systems.

3.6 Interactor Models

This section reviews a group of formal models of interactive systems collectively referred to as *interactor* models. They are more concrete than the models discussed so far, in that they introduce more structure to the specification by describing an interactive system as a composition of independent entities. Interactors are unitary abstractions used in the description of interactive systems. They can be thought of as software-architectural abstractions similar to objects in object-oriented programming. Definitions vary with their intended use. Faconti [62] defines an interactor as:

> '...an entity of an interactive system capable of reacting to external stimuli; it is capable of both input and output by translating data from a higher level of abstraction to a lower level of abstraction and vice versa.'

Faconti considers the interactor as a conceptualisation of a software component serving the communication between a user and an application. Interactors manage some data, and levels of abstraction characterise this data. An interactor bridges two levels of abstraction of the data that is communicated by the interactors. The user interface is assumed to be a layered composition of such interactors, which mediates between the user and the functional core.

Duke and Harrison [55] define an interactor as

> '...a component in the description of an interactive system that encapsulates a state, the events that manipulate the state and the means by which the state is made perceivable to the user of the system.'

The difference between the two definitions, refers to whether the whole interactive system is modelled or just the user interface system. The distinction is not inherent in the model. It pertains to the question of separation of user interface system and functional core, which was discussed in chapter 2, and it embodies a choice for the scope of the interaction design problem.

The term *interactor* has been used also to refer to implementation constructs, e.g. for the input model of the GARNET user interface development environment [136, 137]. This thesis is concerned with formal abstractions of the architectural constructs discussed in chapter 2 and the term is used in this narrower sense below.

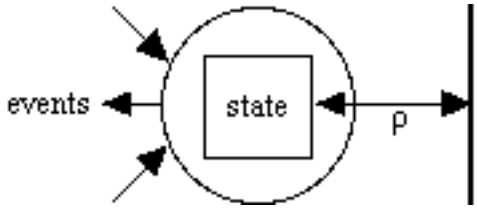


Figure 3.4. Illustration of the York state-based interactor model (adapted from [55]).

The above two definitions correspond to two interactor models, or rather, two families of interactor models. For brevity they are referred to as the York and the Pisa models pertaining to the institution from which they originate. The border between abstract models and interactor models is blurred, partly because they can both be applied at different levels of abstraction. In fact an early version of the York model [55] is only a slight variation of the interactive processes of Sufrin and He. The aim of the York approach has been to formalise abstract properties of interaction and to experiment with expressions of these properties in variations of their interactor model. They do not aim to provide a notation for the specification of user interfaces, as they recognise that other specification languages than Z would be better suited for the representation of the concepts they deal with [55, 59]. On the contrary, the Agents model of section 3.5 aspired to develop a design notation too. In short, the interactive processes and the Agents models could both be classified as interactor models. They are discussed separately, mainly for historical reasons, as the term 'interactor' has been associated with the work discussed in this section.

3.6.1 The York interactor model

In [55], Duke and Harrison propose an interactor model that is a slight variation of the interactive processes of Sufrin and He. They explore the link between the general concept of an object in the object-oriented programming sense and the interactor. The interactor is distinguished from a generic object by the introduction of a rendering function ρ. This provides the environment with a representation of the interactor's internal state (figure 3.4). A formal model of an object as a state-based process is proposed and it is shown how interactors can be seen as a composition of two such objects, where the state of one object describes the display of the interactor and the second describes its inner state. The use of the model in realistic-scale applications is reported in [57, 58].

A small set of composition operators were defined in [55]. They correspond to process algebraic operators like synchronisation, hiding and renaming. An interesting aspect of [55] is their attempt to define a theory of the composition of interactors and, in particular, the composition of their state specifications. However, the meaning and the validity of these compositions is not very clear and subsequent publications have not progressed this work any further. Although the composition operators are quite distinct to those discussed in the context of the Agent model [1], they also attempt to describe formally the composition of interface objects.

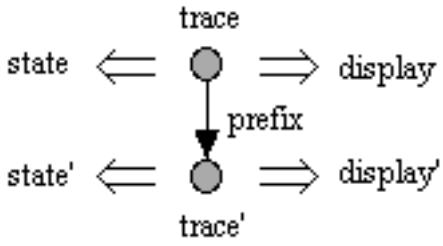


Figure 3.5. The York interactor relates traces of events to their effects on its state. The display is a projection on this state (adapted from [54]).

In [59], Duke and Harrison depart from the use of state-based processes and use a purely event-based abstraction. Instead of the notion of a trace which is founded on a total ordering over events, they represent the observable interactive behaviours by partially ordered sets of (instantaneous) events, called *posets*. An interactor is modelled as a prefix-closed set of posets that characterise the interactions between a user and a system. This model allows for true concurrency in the occurrence of events as opposed to the interleaved concurrency of other interactor models. True concurrency is useful in modelling multi-modal interaction. Another interesting feature of the model is that it allows for the refinement of the event specifications, by mapping an action specification in one poset to a poset. An interactor specification can be studied at increasing levels of detail, related by the refinement relation.

To discuss expression of interaction properties of the GUEPS genre, it is essential to represent the concept of system states and renderings. This later version of the York interactor model is purely event-based so there is no explicit representation of state or rendering. Instead subsets of the action set of the interactor are considered to represent a state or a rendering. The result and view mappings of the state-display model are now defined as a projection of a set of states through a poset of the interactor. The comparison of the internal and the external behaviours of an interactive system can now be defined as relations between posets and it is straight forward to make a classification of interaction properties, similar to the one originally proposed by Sufrin and He [169]. A very similar approach was taken independently in [123] and is discussed extensively in section 7.1, so a detailed discussion on the specification of interaction properties is deferred.

The York interactor model continues the tradition of the PIE model that models interfaces by relating input sequences to their effects through an interpretation function. This is illustrated in figure 3.5. A circle represents some observed behaviour of system, expressed as an ordered set of events. The double-line arrows represent projections that relate a trace to the internal state and the display of a system. Traces may be related by a prefix ordering (single lines in figure 3.5). The York interactor model, like the host of abstract models that preceded it, does not cater well for modelling direct manipulation. As with the PIE model from which it originates largely, it is very much uni-directional and fails to explicate how the interpretation of the input depends on the current contents of the display.

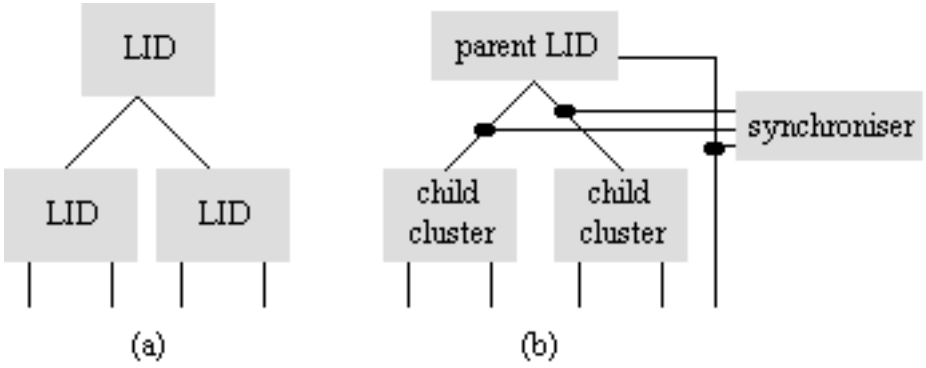


Figure 3.6. (a) Hierarchical composition of LIDs as in [53] and (b) the recursive composition of clusters of [66]. An LID may have many input gates (lines at the bottom side) and a single output gate (lines at the top). Blobs indicate synchronisation of all connected processes.

Recent developments to the state-based version of the York model have introduced the decoration of interactor specifications to indicate the modality of the presentation (e.g. [60] outlines the specification of a flight information system where speech and gesture modalities are combined). These decorations are based upon a theory of presentations, outlined in [56], which aims to describe with precision user perceivable structures of a system's presentation. This extension of interactor specification paves the way for the *syndetic modelling* of interactive systems [63]. Syndetic modelling is a specification technique which allows the description of both the device and the cognitive resources required in interaction to be captured in a single representation. The predictions of the cognitive activity are based upon a model of user cognition [11, 13]. Interactor specifications are combined with a formal model of user behaviour, within a unified framework. The interactor specifications are written mostly in a hybrid notation that uses VDM or Z for the description of the state components and modal action logic for the description of the temporal behaviour of the interactor.

3.6.2 The GKS input model and the Pisa interactor model

The Pisa model emphasises the nature of an interactor as a communication component that supports some part of the bi-directional data flow between user and application. The origins of this approach can be found in attempts to formalise the graphical input model of GKS (graphical kernel system). In [52], CSP is used to model *logical input devices* (LIDS). A logical input device (LID) is an abstraction of a physical input device which describes the manipulation of a specific input data type by a graphics system. This use of CSP for the specification of interaction differs from Squeak [33] or SPI [8] in that it supports a standard architectural structure for the description of graphical interaction. This structure allows a consistent specification technique to be applied for all LIDs and specifications to be created by simple modifications of a set of templates. An interesting suggestion made by Duce et al. in [52] is that if a similar approach would apply to

modelling the output pipeline for graphics, this would enable the description of the interaction between input and output. Observing that the structure used to describe logical input devices could be used to describe input devices at any level of abstraction, a generalisation of the original model was proposed in [53]. The specification of complex input devices could now be achieved out of the hierarchical composition of logical input devices, where one device may provide input to another at a higher level of abstraction.

A first attempt towards a symmetric treatment of input and output, with the aim of formally specifying interactive graphics programs, was reported by Faconti and Paternó in [65], who introduced the term interactor in the context of the formal specification of input/output objects. They used an extension of Hoare's CSP, called ECSP which allowed for the dynamic allocation of statically defined communication channels. Each has an input part that builds up the input value (the measure in GKS parlance) and an output part that stores the corresponding pictures and provides feedback.

The formal specifications of the GKS input model of [53, 65 and 66] identified limitations of the informal description of GKS that still persisted well into a decade in its development. Paternó and Faconti proposed improvements to the input model and specified them in the formal specification language LOTOS [66]. They introduced the notion of a *cluster*, which is recursively defined as an LID or a composition of clusters and a parent LID (see figure 3.6). The three operating modes of the GKS input model were specified in a *synchroniser* process that controls the communication between the children-clusters and the parent. The controller component defined in chapter 4 and more abstractly in chapter 6 is a generalisation of these synchronisers. The specification in LOTOS of an LID in [66] was developed in an effort to improve the GKS reference model. The work that followed it, and that is referred to in this thesis as the 'Pisa interactor model', addressed the more general problem of specifying user interface software.

The user interface system is thought of as a collection of communicating entities, the interactors, that operate concurrently. At any instance the interface may be described as a composition of interactors into a graph that describes the flow of data between them. A Pisa interactor mediates between two data-abstraction levels, raising the level of abstraction of user input and refining the output descriptions it receives. Paternó and Faconti [150] call them transformation abstraction levels, adopting the five abstraction levels distinguished by the Computer Graphics Reference Model. It has to be pointed out though that this is not inherent to the interactor model. User interface software is not usually structured in these five layers and since they do not impact the definition of the interactor they are ignored in the discussion that follows.

As with the York model, various versions of the model have been published that differ slightly. It is rewarding to have a closer look at two versions of the interactor specification: an *extensional* (black box) view which abstracts away from the internal structure and the data operations and an *intensional* (white box) view which proposes a particular internal organisation. Historically the white box view predates the black box

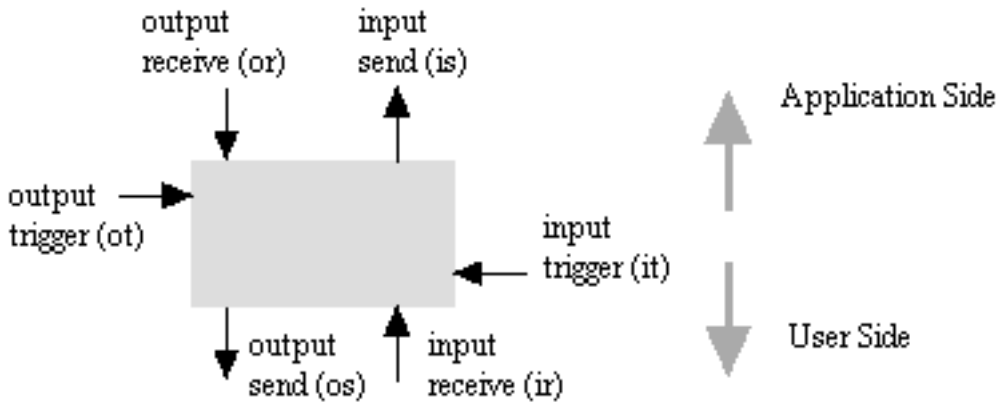


Figure 3.7. Black box view of the Pisa interactor.

view and inherits much from the formalisation in LOTOS of the GKS input model. For the purposes of comprehensibility they are presented in the reverse order.

Extensional Description of the Pisa Interactor

The interactor mediates between two layers of abstraction. The lower level is called the user side and the higher level is called the application side (see figure 3.7). The interactor may receive input from the user side on gate *ir* or from the application side on gate *or*. The interactor accumulates this input, although this is not actually described in the basic LOTOS specification of [54] and uses its current value to update the output data. Output can occur over gate *os* towards the user and gate *is* towards the application. The interactor will construct internally the value that it will output in either direction. The output-send *os* and the input-send *is* are triggered by an event on the corresponding trigger gate.

In basic LOTOS the interactor specification would be simply (adapted from [54]):

```
process interactor [ir, it, is, or, ot , os] : noexit :=
     or;        interactor [ir, it, is, or, ot , os]
[]   ot; os;    interactor [ir, it, is, or, ot , os]
[]   ir;        interactor [ir, it, is, or, ot , os]
[]   it; is;    interactor [ir, it, is, or, ot , os]
endproc
```

Intensional Description of the Pisa Interactor

The intensional description of the Pisa interactor [150] provides more information as to how it may be built. Its structure follows from its origin as an abstraction of input devices for the GKS model. The interactor is formed by the synchronous composition of four processes (figure 3.8):

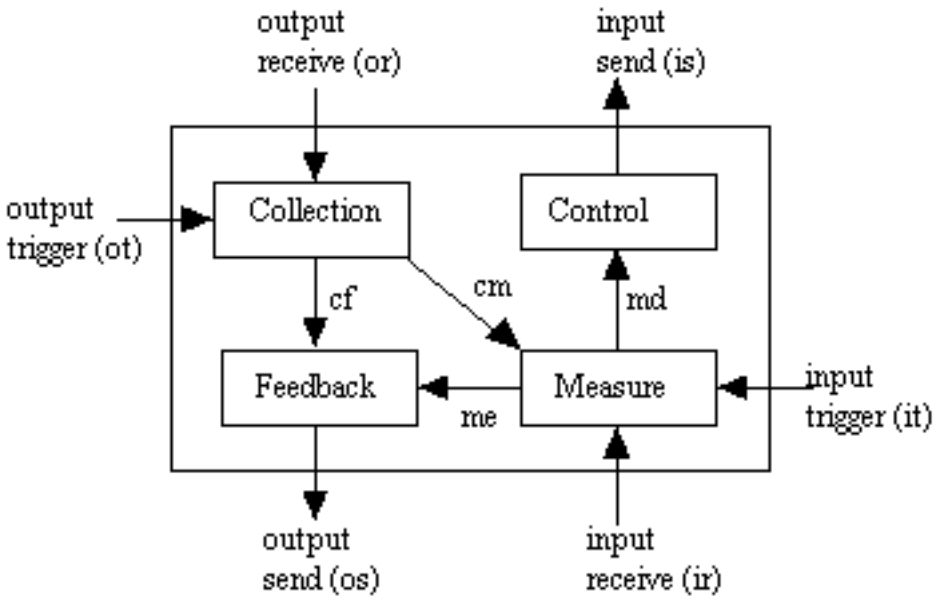


Figure 3.8. 'White box' structure of the Pisa interactor (adapted from [150]).

- The *collection* maintains an abstract representation of the external appearance of the interactor.
- The *feedback* maintains the graphic primitives, i.e. a lower level and more detailed description of the output state. This also captures the intermediate output which results, e.g. as a feedback to user input.
- The *measure* accumulates and interprets input from the user and produces input data of a higher level of abstraction (n+1).
- The *control* delivers the produced data to other interactors or the application itself.

In the general case an interactor has an input and an output behaviour. An interactor that only has a collection and a feedback process, is an output-only interactor. One that does not have a collection component is an input-only interactor, although its feedback process may provide some simple echo to the user.

The data operations of the interactor are described by two data types: the *collection* which is a higher level description of the data and the *picture* which is a lower level description of how this data is displayed. These data types are defined individually for each interactor. Generally three operations are applied to a collection :

- *interpret*. This operation is applied to the input received from the application side to update the collection.
- *trav_meas*. This operation is applied to the *collection* and the result is sent to the measure process.

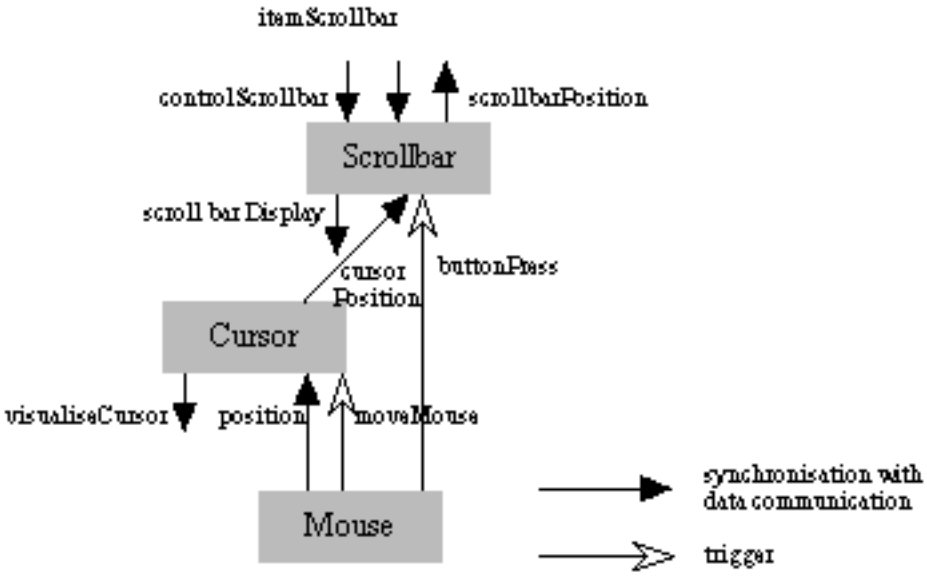


Figure 3.9. Modelling a scrollbar with the Pisa interactor (adapted from [150]).

- *trav_feed*. This operation is applied to the *collection* and the result is sent to the feedback process.

The data type *picture* defines in general an operation *pick* that interprets input data from the user side to detect primitives which will be highlighted by the application of operation *highlight* on the picture primitive. An operation *meas* defines the function that the interactor applies to the input data. The formal specification of the Pisa interactor is outside the scope of this presentation. The reader who wishes to draw comparisons with the ADC interactor model presented in the next chapter is referred to [150] for an extensive description of the Pisa interactor model. Note, that the two versions of the model described here are not observationally equivalent (appendix A.1) , so the two versions of the Pisa interactor are better thought of as two distinct formalisations of the interactor concept.

The internal structure of the Pisa interactor model and the definition of the data types are inherited from GKS. In the context of the present thesis, where there is no commitment to a particular implementation architecture, there is no need to support this internal structure for the interactor. Interface descriptions can be couched in these terms but not necessarily so. In chapter 4, an extensional description reminiscent of the black box view of the Pisa interactor is adopted and extended. Also more meaningful abstractions for the data handling function of the interactor are discussed that borrow from the ideas of the York interactor model.

To specify an entire UIS, using the intensional version of the Pisa interactor, the system is first represented as a group of interactors. The behaviour of each interactor is standard up to the renaming of the gates by instantiation. The specification of the interactor boils down to the specification of the *collection* and the *picture* data types. The interface is

formed by a communication graph whose nodes are the interactors and whose edges represent communication between them. For example, the interaction with a scrollbar is modelled as the graph of figure 3.9. Note that the graph does not have to be strictly hierarchical as was the case in figure 3.6.

The Pisa model potentially facilitates the task of specifying a graphical interface. It provides a conceptual framework for the specifier, but also reduces the task of specification to defining the data types managed by each class of interactors and to the composition of the interactors. The temporal behaviour of each interactor follows directly from the general template. The composition is better described in a diagrammatic form as was argued in [64]. The specification of interfaces in LOTOS is supported by software tools. Standard model checking tools for LOTOS specifications were used in [152,153] to verify properties of user interface specifications expressed in a type of temporal logic. This work is discussed more extensively in chapter 7.

Clearly a design notation for user interaction should not be confined to the behaviours defined within the GKS input model. With this purpose in mind, an intensional description of pre-defined behaviours is a drawback of the model. The internal structure of the model seems excessive for the task of interface design which views interactors as unitary abstractions. It further complicates the definition of the data types with operations that model data transformations for the purposes of communication internal to the interactor. Arguably, simpler specifications will result if only operations on data observable to the environment of the interactor are modelled.

Concerning the composition of interactors, it is clear that to define different modes of operation or to define the communication constraints between interactors, it is necessary to add components like the ‘synchronisers’ of figure 3.6. For example, consider the cursor and the scrollbar receiving input from the mouse. Do they receive the mouse position at the same time? Should the mouse position be sent to one or the other? Should it be sent to both but independently without them having to synchronise? This suggests first that the connections between interactors have to be studied themselves.

A refinement of the Pisa model has been used as the basis of a user interface toolkit for the implementation of user interfaces [154]. This is interesting on its own right as an approach to constructing a user interface development aid. It puts emphasis on the semantic operation of the objects from which the user interface is constructed and not just their presentation. Interactors were categorised on the basis of the type of data they send to the application and whether or not they need to communicate to the application to perform their operation. Interactors provide a conceptual framework that structures the design space. An important contribution of the construction of the toolset is that it validates interactors as an abstraction for user interface software.

The Pisa interactor model has been put to several uses as an aid in interface design. For example, [152, 153 and 155] report reverse engineering case studies of its use. These case studies demonstrate an approach to the analytical evaluation of an interactive system with relation to a task specification.

A more abstract study of the use of interactors by Paternó [149], concentrated on the role of interactors in supporting the data flow between a user and an application. The interface software is modelled independently of any notation as a data flow graph whose nodes are interactors. Static checks of the correctness and the validity of the graph stem from the observation that such data communication is directed and typed. For example, an interface graph can be checked to verify that the data sent from one interactor and received by another are consistent. This can be checked along paths starting from an interactor the user interacts with directly and reaching one which provides input directly to the application. This more abstract version of the Pisa interactor is compositional [149], i.e. a composition of interactors is also an interactor. Unfortunately, this only applies to this abstract level of description. The concrete specifications of the model do not have this property. This point is particularly interesting for the thesis. The interactor model proposed in the next chapter is argued to have this compositionality property as well as providing a practical scheme for the specification of user interfaces as well.

The lessons learnt from the study of the Pisa model are summarised below.

- Interactors are communication entities that interpret and forward data either on the user side or on the application side.
- Interactors are formally specified by instantiating a syntactic template in the LOTOS language. The specification of a user interface reduces to the definition of the data types managed by the interactor and to the composition of interactor specifications.
- Interactors may be used as the basis of a user interface design tool and as a conceptual framework for user interface design.
- The interactor model can be used analytically, e.g. by a static data flow analysis of the composition graph or analysis of its dynamic properties with temporal logic.

3.7 A formal framework for modelling interface software

The review of the approaches to specifying and implementing interactive systems, presented in this and the previous chapter, suggests the salience of events as a concept for modelling interaction. Also evident in this research is the need to have a component of the notation that is particularly suited for the specification of the functionality rather than the behaviour of the interface. Rather than developing a purpose-specific hybrid specification notation, it is wiser to adopt an existing and mature formalism, with a wide user base, a well understood and documented theory and tool support. Accordingly, as mentioned in section 3.1, the LOTOS formal specification language [100, 178] has been adopted for the research presented in this thesis.

The underlying semantic model for LOTOS specifications is an abstract relational model called *Labelled Transition Systems* (LTS). This model underlies many formalisms for the specification of concurrent systems. It is discussed briefly in order to draw attention to the trade-offs it incurs in describing interaction. LTS are discussed rather than

Notation	Meaning
$q \stackrel{\mu_1..\mu_n}{\rightarrow} r$	$\exists q_1...q_n \in Q \mid q \stackrel{\mu_1}{\rightarrow} q_1 \stackrel{\mu_2}{\rightarrow} q_2 \rightarrow ... \rightarrow q_{n-1} \stackrel{\mu_n}{\rightarrow} r$
$q \stackrel{\varepsilon}{\rightarrow} r$	$q \equiv r \quad n \geq 1 \mid q \stackrel{\tau^n}{\rightarrow} r$
$q \stackrel{\mu}{\Rightarrow} r$	$\exists q_1, q_2 \in Q \mid q \stackrel{\varepsilon}{\rightarrow} q_1 \stackrel{\mu}{\rightarrow} q_2 \stackrel{\varepsilon}{\rightarrow} r$
$q \stackrel{\mu_1..\mu_n}{\Rightarrow} r$	$\exists q_1..q_{n-1} \in Q \mid q \stackrel{\mu_1}{\Rightarrow} q_1 \stackrel{\mu_2}{\Rightarrow} ... \stackrel{\mu_{n-1}}{\Rightarrow} q_{n-1} \stackrel{\mu_n}{\Rightarrow} r$
$q \not\stackrel{\mu_1..\mu_n}{\Rightarrow}$	$\not\exists r \in Q \mid q \stackrel{\mu_1..\mu_n}{\Rightarrow} r$
out(q)	$\{\mu \in A \mid \exists r \in Q \cdot q \stackrel{\mu}{\rightarrow} r\}$
Tr(q)	$\{\sigma \in A^* \mid \exists r \in Q \cdot q \stackrel{\sigma}{\Rightarrow} r\}$
Π(q)	$\{<q_1,q_2,...q_n> \mid q_1,q_2,... \in Q \wedge q \stackrel{\mu_1}{\rightarrow} q_1 \stackrel{\mu_2}{\rightarrow} ... \stackrel{\mu_n}{\rightarrow} q_n\}$

Table 3.1. Some notation for describing an LTS.

LOTOS, since they provide the underlying semantic model that determines the expressive and analytical power of the specification language. In other words, the formal specifications of the following chapters could be written or analysed using other formalisms with the same underlying semantic model, as for example Estelle [177]. However, choosing LOTOS as a specification language influences the manner, and ease, with which such systems are specified and the way that architectural concepts are represented.

In the framework of LTS, a system, the user interface in this case, is described using the notions of a global state and indivisible actions that cause state transitions. Each component of the user interface can itself be associated with a corresponding LTS. The class of LTS studied in this thesis can model systems controllable through interactions with their environment, which are therefore considered to be externally observable, and also *internal* or *hidden* actions, which cannot be observed or influenced by an external agent.

Definition. Labelled Transition Systems.

A labelled transition system is a tuple $(Q, A, \stackrel{\mu}{\rightarrow}, q_0)$, where Q is a countable set of states, A is a countable set of elementary actions, $\stackrel{\mu}{\rightarrow}$ is a set of binary relations on Q indexed by $\mu \in A \cup \{\tau\}$, representing transitions between states, and q_0 is the initial state.

A relation $\stackrel{\alpha}{\rightarrow}$, with $\alpha \in A$, describes the execution of an elementary action a. If q_1, $q_2 \in Q$, then $q_1 \stackrel{\alpha}{\rightarrow} q_2$, indicates that when the system is in state q_1 it can perform an action α and reach state q_2. The symbol τ is used to denote a hidden action that effects a 'silent' transition $q_1 \stackrel{\tau}{\rightarrow} q_2$. Table 3.1 summarises some notation which is used in this thesis. It introduces notation for abstracting away from hidden actions and for representing sequences of actions prescribed by the LTS. The most important concepts described by this notation are the set of traces Tr(q) of a labelled transition system and

the set of actions out(q) which may follow a state q Q. Also, an interesting concept is that of a 'path' which is a finite or infinite sequence $q_1,q_2,...$ of states accessible from a state q. The set of paths from a state q is denoted as Π(q). A path is maximal if it is infinite or if it is finite and its final state has no successor state.

LTS already embody some choices for describing interface software adopted for this thesis. For example, only a single action can take place at any time. There is no concept of overlapping actions, of a duration of actions or of true concurrency in their occurrence. Activities are modelled as ordered sets of sequences of actions. With LTS 'parallel' activities are in fact logically interleaved. For the purposes of specifying interface software, it is not clear how useful it is to model true concurrency, or to associate temporal information with actions. It is noted that some of the recent developments of the York interactor model, e.g. [58, 59], have been associated with true concurrent models or modal logics that overcome these limitations of LTSs. Further, a recent report on the Pisa interactor model examined the use of a temporal extension of LOTOS [132].

Adopting LTS as a model for the specification of user interfaces implies that interactive objects, such as those discussed in chapter 2, are thought of as logically concurrent entities. This notion has been identified early on by researchers, as was discussed in the context of object based architectures in chapter 2, and special purpose languages for specifying and programming interaction dialogues, in the beginning of this chapter. In comparison to simpler models like finite state machines, an LTS affords the comparison of systems with respect to the options offered at each step during interaction, rather than the complete sequences of actions that may follow the initial state, before terminating. In this sense, LTS are a more appropriate model for interface software.

The notion of comparing LTS is quite complex. There are many different equivalences between LTS, and it is an ongoing debate which of these notions is most appropriate or 'natural'. This thesis does not attempt to resolve this question. An interesting debate about such notions can be found in [4, 45, 47 and 133] and many others. This debate becomes more relevant to the thesis when user-oriented properties of interface software are discussed in chapter 7. In chapters 6 and 7 comparisons are made between LOTOS behaviour expressions regarding their meaning. These comparisons always pertain to the underlying LTSs. The concepts that are used for these comparisons and the corresponding definitions of equivalences between LTS, are discussed in appendix A.1 at the end of the thesis.

With LTS interaction is represented by the occurrence of actions. The description could apply to any abstraction level, which could range from the superficially abstract, where one action could be 'interact', to the most detailed, and possibly contrived, study of events at a physical and perceptual level, 'lift finger', 'press', 'hear the click of the mouse', etc. Such descriptions may themselves be of interest: e.g. to specify physical interaction devices [5], to classify them [31] or to provide an account of some activity of a very large scale, e.g. [104].

3.8 A brief introduction to LOTOS

This section introduces some basic concepts of LOTOS which are used in the following chapters. It is not a self-contained tutorial. The reader is referred to [157] for a tutorial of LOTOS aimed at practitioners and to [116] where LOTOS is introduced via simple examples. A more theoretically oriented introduction is that of Bolognesi and Brinksma [18]. This section aims to explain the notation so as to facilitate the reading of the thesis. First, a subset of LOTOS, called Basic-LOTOS, is introduced. Basic LOTOS is a process algebra whose actions do not involve data exchanges. An action in this process algebra is denoted by a simple identifier. Full LOTOS extends this process algebra with constructs for the specification of data, operations on the data, and means for communicating data among processes.

3.8.1 Action prefix

LOTOS describes the temporal ordering of *action occurrences*, for which the term *interaction* and *event* will also be used. The *action prefix* operator, written as a semi-colon, expresses the sequential composition of actions. Thus, the expression *a;P* denotes a behaviour where, after interaction *a* the subsequent behaviour is described by *P*.

The simplest behaviour expressions are built into the language and concern termination. Behaviour *stop* denotes inaction, i.e. a behaviour that offers no action, while *exit* denotes a successful termination of a process. For example, an interface that can perform action *interact* and then terminate successfully will be described by the behaviour expression:

```
interact; exit
```

3.8.2 Process definition

LOTOS specifications are structured in process definitions. The syntax for defining a process *interface* with the behaviour described above is as follows:

```
process interface[interact] : exit :=
    interact; exit
endproc
```

In this section, keywords are underlined in the examples. This practice is not continued for the specification segments of the later chapters. On the first line, the set of *gates* of the process is listed between square brackets. Gates are an architectural concept of LOTOS. Actions are ‘observed’ on the gates of the process. In basic LOTOS there is no difference between an action and a gate identifier; this distinction is relevant in full LOTOS where an action occurrence involves the communication of data over a gate. In the header of this process definition, the keyword *exit* denotes that the process should terminate successfully. This, in LOTOS terminology, is the *functionality* of the process which can be either *exit* or *noexit*. Functionality *noexit* means that the process eventually stops or may never terminate, e.g. because of recursion.

3.8.3 Process instantiation

A behaviour expression may involve a process instantiation. In the process instantiation the *formal* gate identifiers are renamed by the *actual* gate parameters. For example, the process instantiation

```
interface [graphicalInteraction]
```

describes the behaviour

```
graphicalInteraction; exit
```

Process instantiation is useful in defining recursive processes. An interface that does not terminate after a single interaction will be described as:

```
process interfaceForEver[interact]: noexit :=
    interact; interfaceForEver[interact]
endproc
```

Note, how the functionality is now *noexit*. In chapter 4 interactors are defined as non terminating processes unless special constructs are added to specify the interruption of such recursive behaviours.

3.8.4 Choice

Process instantiations may be composed to construct more complex behaviour expressions. LOTOS provides a set of process composition operators. The action prefix operator has been introduced already. Alternative behaviours may be associated with the *choice operator*, denoted as []. Consider for example a user who will either think or interact. This behaviour is modelled by a LOTOS process as follows:

```
process user [interact, think] : exit :=
    think; exit
[]  interact; exit
endproc
```

The choice operator could, of course, relate any two behaviour expressions. For example it could be that a system offers two different modalities for interaction, graphical interaction and speech interaction. Using the process *interface*, this 'multi-modal' system could be described by the behaviour expression:

```
    interface [graphicalInteraction]
[]  interface [speechInteraction]
```

LOTOS supports a generalisation of the choice construct. One form of this generalised choice which is used extensively in the thesis, is used below to describe the same 'multi-modal' system.

```
choice g in [graphicalInteraction, speechInteraction]
    [] interface[g]
```

This construct is useful for expressing a choice among many instances of a process. For each instance of the process, the formal gate *g* is actualised with an element of the gate list associated with the generalised choice construct.

3.8.5 Synchronisation and interleaving

The most commonly used operator, in this thesis, is the *synchronous composition operator*. The process *interface* defined above, may synchronise with other processes on its gate *interact.* Consider for example a user modelled as follows:

```
process user [interact, think]: exit :=
    think; interact; exit
endproc
```

Here, the user is modelled as a process that may engage in an action *think*, then *interact* and terminate successfully. From these impoverished descriptions of an interface and a user, a simplistic model of interaction may be obtained as the synchronous composition of two process instantiations. This is denoted using the *synchronous composition* operator |[...]| as follows:

```
process interaction[graphicalInteraction, think]:exit:=
    user[graphicalInteraction, think]
            |[graphicalInteraction]|
    interface[graphicalInteraction]
endproc
```

The two processes synchronise over the gates listed between the square brackets. The two processes specify independent behaviours that have to synchronise, to rendezvous, on the occurrence of an action on the synchronisation gate, i.e. *graphicalInteraction* in this example. The synchronisation of the two processes above describes the behaviour:

```
think; graphicalInteraction; exit
```

In the above case, the two processes synchronise on all their common gates. A shorthand for the special case where the two processes synchronise on the union of their gates is to omit the gate set between the vertical bars. This is called *full synchronisation*. A rather problematic interaction could be modelled by the full synchronisation of a user and an interface:

```
    user[graphicalInteraction, think]
||  interface[graphicalInteraction]
```

This behaviour expression is equivalent to a deadlock (stop) as the process *interface* can not match an interaction with the user on gate *think*.

To specify two processes operating concurrently, but which do not synchronise on any of their gates, the *interleaving* operator ||| can be used. For example the combined behaviour of two users described by the same process, but having different thoughts

thinkA and *thinkB* and interacting independently with the same graphical interface could be written as:

```
    user[graphicalInteraction, thinkA]
|||  user[graphicalInteraction, thinkB]
```

Their respective thoughts and interactions are not constrained by each other. However, the users might find themselves constrained by having only one interface to work on:

```
(   user[graphicalInteraction, thinkA]
|||  user[graphicalInteraction, thinkB])
||   interface[graphicalInteraction]
```

This behaviour expression could be written using the choice operator as follows:

```
    thinkA; (graphicalInteraction; thinkB; stop [] thinkB; graphicalInteraction; stop)
[]  thinkB; (graphicalInteraction; thinkA; stop [] thinkA; graphicalInteraction; stop)
```

This equivalence between behaviour expressions that use the choice operator and others using one of the parallel operators, i.e. synchronisation, interleaving and full synchronisation, is an instance of the expansion theorem [133]. This theorem captures the essence of what is called an *interleaving semantics*. The parallel execution of two actions a and b is defined as a choice where either a is followed by b or vice versa. It turns out that any LOTOS behaviour expression can be written as a choice between behaviour expressions, each prefixed by a single action. This structure is called the *action prefix form*. Essentially the action prefix form describes the behaviour of a system as a choice between all the possible behaviours that exist for this system.

3.8.6 Enable

When users A and B work in succession, rather than in parallel, their combined behaviour can be described using the *enable* operator of LOTOS, denoted by >>, as follows:

```
user[graphicalInteraction, thinkA] >> user[graphicalInteraction, thinkB]
```

This expression specifies the following sequence of actions

```
thinkA; graphicalInteraction; δ; think; B; graphicalInteraction; exit
```

The enable operator requires that its left operand terminates successfully. This successful termination is signified in the action sequence by the hidden action δ, which is not a LOTOS construct but results from the termination with an *exit*.

3.8.7 Disable

To describe a disruptive user B that may at any time interrupt user A, the *disable* construct will be used. The disable operator is denoted as [>. Consider the behaviour expression:

```
    userA[graphicalInteraction, thinkA]
[>  userB[graphicalInteraction, thinkB]
```

The brief behaviour of user A may be observed in its totality, in which case the whole behaviour expression terminates. However, at any point during the interaction of A, B might interfere. Using the action prefix notation this would now be written as:

```
    thinkA; graphicalInteraction; exit
[]  thinkA; graphicalInteraction; thinkB; graphicalInteraction; exit
[]  thinkA; thinkB; graphicalInteraction; exit
[]  thinkB; graphicalInteraction; exit
```

Each 'leg' of this behaviour expression describes a particular 'point' at which B interrupts the behaviour of A.

3.8.8 Hide

The system behaviour may not always be possible to observe externally. This is specified by designating actions to be 'hidden' from the environment of the process, using the operator *hide*. For example, the thoughts of a user are normally not observed externally. A more accurate description of how the user appears to the observer would be:

```
hide think in user[graphicalInteraction, think]
```

To the environment, the user's thought action is a hidden, and therefore silent, action. LOTOS uses the letter *i* to denote non-deterministic hidden interactions, where their (hidden) occurrence is explicitly specified. Note, that *i* is different to a hidden success action δ which is not explicitly specified in LOTOS. Using i the last behaviour expression can be written as

```
i; graphicalInteraction
```

The internal event *i* is not observable by the environment. It can occur spontaneously without the participation of the environment.

Hide concludes the introduction to basic LOTOS. In table 3.2 the syntax and operational semantics of basic LOTOS are summarised. The semantics are defined by inference rules regarding the transitions specified. Using these rules a basic LOTOS process can be interpreted as a LTS. Both i and δ are modelled by a τ transition in the underlying LTS interpretation.

3.8.9 Action specification in full LOTOS

An action specification in full LOTOS, apart from a gate identifier, includes a list of *attributes*, which specify the data that is exchanged with the action. LOTOS allows for many data exchanges to occur on a single action. An attribute may be a *value declaration* or a *variable declaration*. In the examples that follow, it is assumed that

Name	Syntax	Axioms and Inference Rules
inaction	stop	
successful termination	exit	exit $\xrightarrow{\delta}$ stop
action prefix	g;B	g;B $\xrightarrow{g}$ B
choice	$B_1[]B_2$	if $B_1 \xrightarrow{\alpha} B_1'$ then $B_1[]B_2 \xrightarrow{\alpha} B_1'$ if $B_2 \xrightarrow{\alpha} B_2'$ then $B_1[]B_2 \xrightarrow{\alpha} B_2'$
enabling	$B_1>>B_2$	if $B_1 \xrightarrow{\alpha} B_1'$, $\alpha \neq \delta$ then $B_1 >> B_2 \xrightarrow{\alpha} B_1' >> B_2$ if $B_1 \xrightarrow{\delta}$ stop then $B_1 >> B_2 \xrightarrow{i} B_2$
disabling	$B_1[>B_2$	if $B_1 \xrightarrow{\alpha} B_1'$, $\alpha \neq \delta$ then $B_1[> B_2 \xrightarrow{\alpha} B_1'[> B_2$ if $B_1 \xrightarrow{\delta} B_1'$ then $B_1[> B_2 \xrightarrow{\delta} B_1'$ if $B_2 \xrightarrow{\alpha} B_2'$ then $B_1[> B_2 \xrightarrow{\alpha} B_2'$
hiding	hide G in B	if B $\xrightarrow{\alpha}$ B′, α G then hide G in B $\xrightarrow{i}$ hide G in B′ if B $\xrightarrow{\alpha}$ B′, α G then hide G in B $\xrightarrow{\alpha}$ hide G in B′
renaming	B[H]	if B $\xrightarrow{\alpha}$ B then B[H] $\xrightarrow{H(\alpha)}$ B′[H]
parallel composition	$B_1\|[G]\|B_2$	if $B_1 \xrightarrow{\alpha} B_1'$, α G $\{\delta\}$ then $B_1'\|[G]\|B_2$ if $B_2 \xrightarrow{\alpha} B_2'$, α G $\{\delta\}$ then $B_1\|[G]\|B_2' \xrightarrow{\alpha} B_1\|[G]\|B_2'$ if $B_1 \xrightarrow{\alpha} B_1'$, $B_2 \xrightarrow{\alpha} B_2'$, α G $\{\delta\}$ then $B_1'\|[G]\|B_2' \xrightarrow{\alpha} B_1'\|[G]\|B_2'$
process instantiation	P	if P:= B, B $\xrightarrow{\alpha}$ B′ then P $\xrightarrow{\alpha}$ B′

Table 3.2. LOTOS syntax and semantics (adapted from [28]).

entities such as natural numbers, points, and colours have been defined. Thus 0,1,2,... will be natural numbers, (0,0) will be a point, red will be a colour, etc. These data types may be defined using the ACT-ONE component of LOTOS discussed later. For the moment their definitions are not so important.

A value declaration has the form !E where E is an ACT-ONE expression that describes a data value. For example, !0, !(0+1) !red and !(red+white) could be value declarations. Assuming some sensible definitions for the operator +, these could specify the actions g!0, g!1, g!red and g!pink respectively.

A variable declaration has the form ?x:t where x is a name of a variable and t is its sort identifier. For example, g?x:nat specifies any of the actions g!0, g!1, etc., and g?colour could be g!red, g!blue, etc. Action specifications may be associated with a *selection predicate* restricting the range of values specified by a value expression. For example, g:x?nat [x<3], specifies any of the actions g!1, g!2.

3.8.10 Interprocess communication in LOTOS

A set of values may be communicated from one process to another when the lists of attributes associated with their respective action specification agree on a single tuple of values. Consider the processes *user* and *interface* discussed in the previous section. A contrived description of the user would be that the user thinks of *aThought* and subsequently points at the middle of the screen. The interface reads a point on its gate *interact*.

```
process user[interact, think] : exit :=
    think!:aThought; interact!middleOfTheScreen; exit
endproc

process interface[interact] : exit :=
    interact?x:aPoint; exit
endproc
```

The behaviour expression below will specify the input of a point, namely the middle of the screen.

```
user[graphicalInteraction, think]
    |[graphicalInteraction]|
interface[graphicalInteraction]
```

In fact, this last behaviour expression could be written as:

```
think!aThought; graphicalInteraction!middleOfTheScreen; exit
```

3.8.11 State parameters for processes

A process may be associated with some local state parameters. These are declared between parentheses after the list of gates. Returning for a moment to the recursive process *interfaceForEver*, let its state consist of a single point. The process definition would be:

```
process interfaceForEver[interact](x:aPoint) : noexit :=
    interact?y:aPoint; interfaceForEver[interact](y)
endproc
```

Such a process, when synchronising with the user, as in the previous paragraph, will instantiate itself recursively as:

```
interfaceForEver[interact](middleOfTheScreen)
```

This specification of interaction describes an infinite repetition of the following sequence of interactions:

```
think!aThought; graphicalInteraction!middleOfTheScreen
```

Table 3.3 is adapted from [18]. It lists the possible interactions between two processes A and B that synchronise over a gate g.

process A	process B	synchronisation condition	type of interaction	effect
g!E1	g!E2	value(E1)= value(E2)	value matching	synchronisation
g!E	g?x:t	value(E) is of sort t	value passing	after synchronisation x = value(E)
g?x:t	g?y:u	t=u	value generation	after synchronisation x=y=v where v is some value of sort t

Table 3.3. Types of interaction between synchronised LOTOS processes.

3.8.12 ACT-ONE data types

ACT-ONE specifications are structured into type definitions. These group together descriptions of the sets of the data values, called *sorts*, the operations upon them, and the properties of these operations defined in an equational framework. A data type will include a set of *equations* which specify which expressions are considered equal. There are no data types built into the language. However, some standard libraries of abstract data types are part of the LOTOS standard [100]. These provide definitions of the most commonly used data types, e.g. Boolean, Naturals, Sets, etc.

The data type *point* below uses most of the constructs of ACT-ONE that are used in the specifications of this thesis.

```
1   type point2D is NaturalNumber
2   sorts aPoint
3   opns
4           middleOfTheScreen:          ->      aPoint
5           mkPoint:Nat,Nat             ->      aPoint
6           xcoord:  aPoint             ->      Nat
7           ycoord:  aPoint             ->      Nat
8           setX:    aPoint, Nat        ->      aPoint
9           setY:    aPoint, Nat        ->      aPoint
10  eqns
11  forall p:aPoint, n,m:Nat
12  ofsort Nat
13          xcoord(middleOfTheScreen)=0;
14          ycoord(middleOfTheScreen)=0;
15          xcoord(mkPoint(n,m))=n;
16          ycoord(mkPoint(n,m))=m;
17          xcoord(setX(p,m))=m;
18          ycoord(setY(p,m))=m;
19          xcoord(setY(p,m))=xcoord(p);
20          ycoord(setX(p,m))=ycoord(p);
21  endtype
```

In line 1 of this specification the data type *point2D* is designated to *extend* the data type *NaturalNumber*. Extension of a data type with new sorts, operations and equations is a very simple way of reusing existing data types. Instead of re-using only one data type,

further data types could be listed after *NaturalNumber*. In this case these data types would be *combined*. Other ways of data type re-use, i.e. renaming, parameterisation and actualisation, are discussed in ACT-ONE tutorials, e.g. [44, 61].

In line 2, the sorts of the data type are listed. This data type has only one sort (*aPoint*) apart from those of data type *NaturalNumber*. Lines 4 to 9 define the syntax of operations, i.e. their arity and the sorts of data involved. Operation *middleOfTheScreen* is an example of how constants are defined as operations without arguments. Line 6 and 7 define two *inquiry* operators. Applied to an expression of type point they return a natural number which is one of its coordinates. Operation *mkPoint* is a constructor operator, which builds up a complex value from simpler ones. *setX* and *setY* are examples of *transformer* operators, which modify the coordinates of a point. The sorts and the operations of a data type define its *signature*, i.e. its syntax. Usually a subset of the constructor and transformer operators is sufficient to generate all the possible terms which follow this syntax. The remaining constructors and transformers may be rewritten as expressions which involve only this subset of operators.

The equations define equalities between expressions which, in the example, correspond to the properties of the two dimensional points. For example the middle of the screen has coordinates (0,0), which makes it a strange screen but an easier one to specify. Note how in this data type inquiry operators are applied to all constructor and transformer expressions. This helps write *sufficiently complete* specifications of the data types [115]. Equations should be written for each application of an inquiry operator to all the constructors and transformers which generate the terms of the data type. In the examples of chapter 4, completeness of this kind is not a predominant concern. However, in practical situations, striving for complete data types is commendable.

This exposition has been very economical. It introduces only a few simple elements of the language used in the thesis. A tutorial directed at practitioners who wish to use ACT-ONE as it is supported by LOTOS can be found in [157]. An extensive introduction to algebraic specifications using the ACT-ONE language can be found in [61] and a brief overview in [44].

3.8.13 Specification styles for LOTOS

In the opening of this chapter it was stressed how it is necessary to structure specifications in order to manage their size and complexity. This is a general requirement for using formal specifications [73]. In section 3.4 an example of a structuring for CSP specifications by Alexander was discussed. This section reviews some results from research in the use of LOTOS for the formal specification of international standards. Researchers in this area faced the problem of making their specifications easy to understand by a wide audience and used by large teams of developers. Their response was to develop a set of standard *specification techniques* or *styles*.

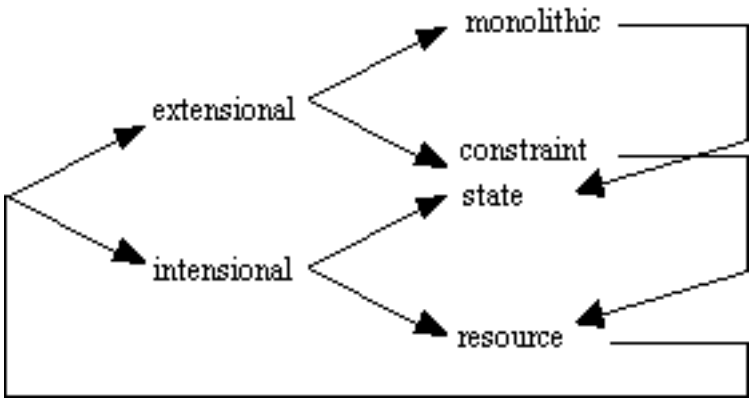


Figure 3.10. Formal software development as a transition between specification styles in LOTOS (adapted from [180]). Arrows indicate transformations of the specification.

A *specification style* is an approach to structure specifications in a given specification language. It is a conceptual guide for the specifier that determines the kind of concepts used to describe a system, e.g. constraints, states and transition between states, objects, etc., and their relationship to language constructs. Different specification styles are appropriate for different purposes in the process of designing some software. Instead of selecting a specification style on the grounds of personal preferences and aversions, it is advisable that authors and readers of specifications be confronted with a standard set of styles [180]. A standard specification style, or set of styles, enhances the homogeneity of a specification. Arguably, this encourages the development of implementation conventions and fosters faster implementations.

Figure 3.10 illustrates how specification styles, listed at the right side of the figure, may serve different purposes. The purpose of a specification in this context, refers to what it aims to describe. An *intensional description* supports the specification of the architecture of a system. An *extensional description* is a more abstract description that specifies what the system does without reference to its internal structure. The four specification styles shown in figure 3.10 are defined in terms of how they structure the specification, what they try to describe and what language constructs they use.

- The *monolithic style* models the system behaviour as alternative sequences of observable interactions in branching time. It prohibits the use of parallel operators.
- A *constraint oriented* specification models externally observable actions of a system. Their temporal ordering is defined by a conjunction of different constraints. No hiding may be used.
- A *state oriented* specification will be more or less like a state transition network specified in LOTOS. No parallel operators are used.

- The *resource oriented* style models both internal and external interactions. The system is modelled as a set of resources (objects) which are interconnected through gates. All operators are allowed.

The specification styles introduced can be thought of as a conceptual aid for writing specifications. Provided that the reader of a specification recognises it, a particular specification style should help understand the specification. LOTOS is a constructive language, in that it helps build up specifications from smaller scale modules. The constraint oriented style makes it also possible to write specifications very abstractly, practically in terms of declarative assertions about the behaviour of the system specified [176]. In practice a mix of specification styles is required. Choosing what part of the system functionality to specify in the behavioural component of LOTOS and what in terms of the data typing component is also a question of specification style. A relevant discussion can be found in [76] where it is argued that the process algebra can help overcome the difficulty of specifying exceptions and error conditions in an equational language like ACT-ONE. A formal model for interface software, like the interactor model discussed in chapter 4, can be seen as a guide to the specifier as to the appropriate mix of specification styles. In addition, the interactor model provides a mapping of interface architecture concepts to mathematical concepts particular to the LOTOS specification language. Vissers [179] calls this mapping an *architectural semantics*.

3.9 Conclusion

This chapter started with a discussion on the role of formal specifications in the development of user interface software, concluding that they have a potential to help user interface development, without though replacing current methods and practices. It then reviewed some influential research works in the formal specification of user interfaces. Over approximately the last fifteen years that this survey has spanned, there has been a considerable change to the research directions in the area. Partly, this reflects the lessons learnt early on and, partly, the developments in user interface technology. Early attempts at the specification of user interfaces, discussed in section 3.3, were directly related to research in UIMSs and their aim was to find executable notations that would serve for the succinct representation of dialogues. Interaction was very much language based, so original approaches used relevant formalisms. Later on, event based notations were identified as the most appropriate means of specifying these dialogues. These event based notations are hybrid, in the sense that they link an event specification with some specification of data and of operations on the data.

The most significant examples of formal specifications of user interfaces, discussed in section 3.4, aimed to demonstrate and assess the potential benefits of a formal method. Such experiences prompted the development of special purpose specification languages for the domain of user interfaces. Similar to dialogue notations, they seem to focus on the specification of events and their effects, regardless of the formal framework adopted. This can also be observed in the specification of abstract models, discussed in section

3.5, and their evolution to interactor models. With only a few exceptions the interactors are specified using hybrid specification techniques. The thesis has adopted a hybrid formal specification language called LOTOS for the specification of user interface software. Section 3.7 discussed the notion of events supported by LOTOS and section 3.8 introduced briefly the LOTOS language.

Formal models of user interface software have gradually introduced more structure in specifications. This structure is embodied in the definition of formal models of interactive objects called interactors. The interactor concept, as defined in section 3.6, has evolved from an attempt to specify interactive systems abstractly and formally, and also, from research concerned with the formal description of the architecture of graphical interactive software. The introduction of more structure in the formal models is reminiscent of the development of object based architectures for user interface software. This analogy was underlined in section 3.2, where formal specification techniques were characterised with respect to their abstractness and their structure.

The survey narrowed down to the presentation of two families of interactor models, in section 3.7, which are referred to as the York and the Pisa interactor models. Apart from the fact that they are defined in different formal frameworks, they describe very similar abstractions. By now, there is some testimony to the validity of these models and to their potential, as the various citations have shown. The discussion has appraised the two models, and their various versions, trying to draw lessons from them.

The York interactor model seems more abstract and better suited for analytical use. Properties of interaction can be formally expressed in terms of the relationship between abstractions of the state and the display for a particular interactor and the sequence of events that leads up to these states. One of the limitations of the York interactor model noted in this chapter was its limited malleability as a design notation. In chapter 7, the thesis examines how properties of interaction may be expressed within a more constructive framework that facilitates the specification of user interfaces.

On the contrary the Pisa interactor model provides a framework for the specification of interfaces that may reduce the specification effort and has the potential to be scaled up. Various uses of the model in the design cycle were cited and they will be discussed more in subsequent chapters. It was noted how the Pisa interactor model could be simplified by abstracting away from a particular implementation architecture that it has inherited from the GKS input model. This point is revisited in the next chapter.

Attention was drawn to how the models presented approach the issue of the composition of interactor specifications. In most models interactors, or agents, or logical input devices are composed to form more complex specifications. In the state based specification of interactors it was quite challenging to model the semantics of the composition. The process algebraic approaches seem to offer advantages in this respect, although much of the structure that makes interactors meaningful as an abstraction and as a design tool is quickly lost in a composition expression. This problem is discussed extensively in later chapters.

Chapter 4

The ADC interactor model

This chapter introduces the ADC interactor model and presents its formal specification as a LOTOS process. First, a few elementary interactive components are discussed illustrating the requirements from the ADC model as a representation scheme, independently of its formal specification. This reflects the aspiration that the model is not only an aid for writing formal specifications in LOTOS. Rather, it provides a conceptual framework for thinking about user interfaces and their components, which captures essential aspects of their nature. However, the model was developed in LOTOS so it is strongly influenced by the features of the language, e.g. multi-way synchronisation, support for specifying both data handling and event ordering. The exposition below extends that of [123] where the model was originally introduced and incorporates some of the later changes to the model. An example specification of a scrollable list is presented. This example illustrates the use of the model and motivates the theoretical arguments of later chapters.

4.1 Dimensions for the description of interactive components

Some simple examples of hardware and software components are discussed briefly. The examples do not aim to provide an accurate description of these devices, but rather, to illustrate the essential dimensions for describing user interface software and its components. These dimensions are mapped onto the components of the interactor model introduced in the following sections.

The first example concerns old fashioned tape recorders of the 1970s and early 1980s. Most have an array of mechanical buttons which can be thought of as the interface of the apparatus. To play a tape the user has to push down a simple mechanical button with the indication ‘Play’ printed on it or next to it. Normally, this starts playing the tape. If the tape has finished or no tape has been inserted, then the button remains pressed, indicating the command last issued by the user, but the tape recorder plays nothing. The button serves to issue a command to the system and to show its own status to the user.

Playing the tape can be interrupted with the 'Stop' button. In most models, this can not remain in the pressed position and only has an effect when the 'Play' button has been pushed before. In some models the 'Stop' button ejects the tape if 'Play' has not been pressed. This behaviour indicates that the history of the commands issued to the tape recorder is important in interpreting further commands. The user is assisted in predicting the effect of a command by observing the state of the buttons. Also, the user is physically not able to press the 'Play' button twice. These are examples of *dialogue* constraints enforced by the design of the artefact.

Later models that became popular in the mid-1980s introduced a panel of touch sensitive buttons to control such appliances. A drawback of these buttons from the users' point of view is that they do not show their state: they cannot remain pressed. To compensate for this, Light-Emitting-Diode (LED) indicators have been added to the buttons. When a 'Play' button is pressed, conventionally a green arrow pointing to the right lights up giving feedback to the user. This LED-equipped button informs the user about the state of the button and also about the state of the apparatus. A green light means that the tape is actually playing. When the apparatus is playing the user can push the 'Play' button again although this action will have no effect.

In later electronic appliances like compact-disc players, the push buttons lost their green LED lights. A separate LED display was added instead, which provides much more information to the user about the application: it does not just indicate that it is playing, but also, which track and how far in the track play-back has reached. In such appliances the dialogue for the 'Play' button has been modified. It can be used to restart play from the beginning of the current track, allowing the user to issue the play command many times successively.

The 'Play' and 'Stop' buttons, the LED display, and the speakers of the apparatus can be thought of as its interface. They exhibit some of the dimensions adopted to describe interfaces using the ADC interactor model:

- An interactor is used to input commands to the application.
- An interactor is used to output information from the application to the user.
- The interactor is characterised by its own state, part of which it may show to the user directly or through other interactors. The state of the interactor is modified both by user input and by changes to the state of the application.
- The state of the application and the state of the interactor itself may determine what input the user is allowed to issue.
- The user may be prohibited from certain actions because of previous interactions. Similarly user commands may be interpreted differently depending on the previous events.

The last two points are very similar. Whether or not a command is available and how a command will be interpreted may be considered to depend on either the previous history

of interactions or the state of the interactor. This observation merits an explanation before making some last remarks about consumer electronics. The term 'state' is widely used in HCI and it is mostly considered self explanatory. However, it is important to remember that what is considered state and what not is a property of the representation scheme used and not of the specificand, i.e. the interface described. The choice of what to represent as state and what as events is influenced by the purpose and the style of the representation. The duality between models of interaction using histories of events or as a state-based model has already been discussed in the context of the PIE model (chapter 3), following the argument of Dix [49]. Comparisons between specification notations which use different formal frameworks, as for example the comparison between a model based specification, and an event based model reported in [144], or styles of specification [179, 180 and 76] can be seen as instances of the same issue.

This issue is also relevant to the last of the tape recorder components that will be discussed. The volume control has largely remained the same throughout the development of tape recorders, even up to its on-screen equivalent on today's multi-media applications (an example of such a component is specified in chapter 5). In the latter case, the user operates an on screen representation of a potentiometer to alter the value of the volume setting. This value ranges between zero and some maximum value. A potentiometer is different from the push buttons which issue a simple command without a value. In this case, the interface builds up some value which it sends to the application. So one more point is added to the list above:

- The interactor builds up some data which it sends to the application.

Alternatively, changing the volume may be modelled as a choice of events, that correspond to discrete values of the volume. Indeed, some of the older volume controls have discrete steps for the setting of their potentiometer. Again it is a choice for the modeller whether to think of the volume as a value that is being built up by the interface and sent to the application, or as the occurrence of one event out of a finite set. The latter may seem contrived, but it may seem equally contrived to model the 'Play' button as sending a value 'play' to the application. In other words, the data held by the interactor and the events that describe its interactions with the environment are not inherently orthogonal attributes of the specificand. The interactor model should not impose to the designer a scheme for how to model these attributes of the specificand but should provide the appropriate dimensions for their description.

Something that cannot be inferred from the observation of such elementary devices, or even of command-line interfaces, is how user actions are sometimes interpreted on the basis of the display. This issue was raised in section 3.5 as a limitation of functional representations of the PIE model. This argument has been extensively discussed in [174] where the inadequacy of simple functional models for describing direct manipulation was discussed. It is added to the list of requirements for modelling interactors:

- Input by the user may be interpreted on the basis of the display content.

In summary, the user interface can be described as an entity which supports the communication between a user and a functional core. The interface can be described in terms of its interactions with the user and the functional core, their purpose and by their effect on two data representations. One data representation pertains to the *display* of the interface. The second data representation partly determines how the interface is used as a store of data to be conveyed to the functional core. This will be called the *abstraction*. The specification of the temporal ordering of the interactions, is called the *controller* component of the interface.

The requirements listed are not particular to a formal specification of a user interface, although the formal specification scheme presented in the next section takes them into account. In section 5.7 such an 'external' view of the interactor as a communication entity provides a conceptual guide for writing the formal specifications of user interface software.

4.2 Overview of the ADC model

The ADC (Abstraction - Display - Controller) interactor model distinguishes three orthogonal aspects of the interactor, which are described in distinct modules:

- The data held by the interactor and the operations upon the data.
- The link between actions that describe the interactor behaviour and their effect on the data. This is described in the *Abstraction and Display Unit* (ADU).
- The temporal ordering of the behaviour of the interactor. This is described in a process without any local state parameters called the *Controller Unit* (CU).

An ADC interactor holds two kinds of data. These are distinguished by whether they are displayed or not. The *abstraction* is the information managed by the interactor that is not displayed. The *display* is information displayed either directly or indirectly through other interactors. Each interactor specification uses an ACT-ONE abstract data type, the *abstraction-display data type* (AD), which describes these data components. A generic template for the definition of this data type is described in the next section.

The Abstraction and Display Unit (ADU) is a LOTOS process. The abstraction and the display are its local state parameters. The ADU may interact with its environment through its gates. It is only through such interactions that the environment may read or modify the values of the abstraction and the display. Characteristically, the ADU is neutral with respect to the temporal ordering of its interactions. At all times it is ready to interact with its environment at all its gates.

The temporal ordering of these interactions is determined by the Controller Unit (CU) which 'controls' all the gates of the ADU. The CU does not store any data received or constructed by the interactor, and it does not apply any operations to such data. Rather, it only specifies the ordering of interactions on its gates. Its 'dialogue state' refers to

which actions it is ready to offer at any moment and what future interactions this may give rise to. This is quite a distinct notion than the state of the ADU, which is sufficiently described by the values of its local variables. Some standard behaviours are built in the ADC interactor structure. These behaviours have the effect of starting, suspending, resuming, restarting and aborting the operation of the interactor. For short these standard behaviours and the corresponding gates through which they are effected are collectively referred to as SSRRA behaviours (respectively gates). The specification of the relevant behaviours requires only the instantiation of the respective gate identifiers.

Interactors can communicate with other interactors and possibly with the user or the application. The interface may be modelled as a graph whose nodes are the interactors and whose edges correspond to connections between them. Connections are modelled as the synchronisation of two or more interactors over a pair of gates. Interactors may also be connected to agents representing the application and the user. Interactions on the connected gates may involve communication of data, which is directed, or simple 'control' events that have no direction. In an extreme case, but not an unlikely one, the whole interface can be modelled as a single interactor. Indeed, whenever the interface is studied as a whole it will be most useful to model it as a single interactor. One of the motivations of the model, that will be discussed quite thoroughly in chapter 6, is the ability to alternate between different views of the interface as a graph of interactors or as a single interactor. This is an important characteristic of the ADC formal interactor model. The model is applicable for both the parts and the whole of the interface. In both cases, the model concerns an arbitrary level of abstraction which is determined by the granularity of the actions discussed.

A gate of the interactor groups interactions with a similar purpose. Interactions on the same gate of the interactor have a common 'role', i.e. to support input or output of data, or simple synchronisation, to cause the application of predetermined operations on the data representations managed by the interactor. The gates are themselves grouped into three 'sides': the *abstraction side*, the *display side*, and the *controller side* of the interactor. If the whole of the interface is modelled as a single interactor, then its abstraction side pertains to interactions with the application and its display side concerns the graphical interaction with the user.

To make a gradual and more comprehensible presentation of the ADC model, a simplified version that does not incorporate the SSRRA behaviours is presented first. In this version the description of the CU reduces to one of its components the *Constraints Component* (CC). There is much merit to the use of such a simple model as it exhibits most of the properties installed in the ADC model and it is clearly simpler.

4.3 Specification of the Abstraction-Display (AD) data type

The gates of the ADU support communication into or out of the interactor. In figure 4.1, they are shown as thick grey arrows. These gates are assigned either to the abstraction

side or the display interactor. In ADU is rectangle. The rectangle abstraction side side of the display side of the may:

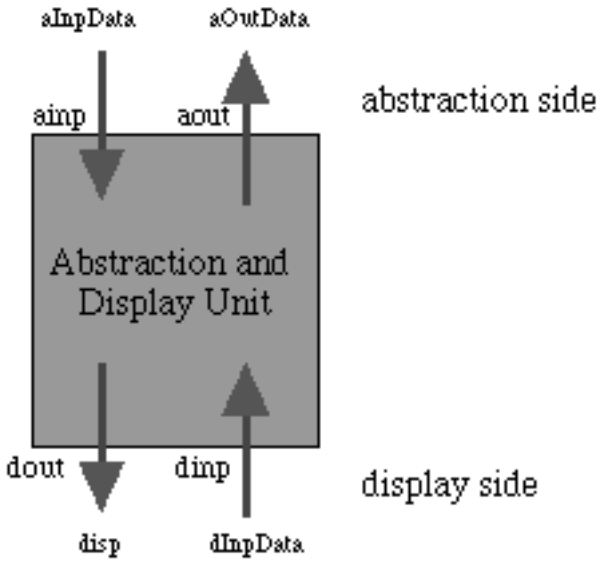


Figure 4.1. The ADU process, its gates and state parameters.

side of the figure 4.1 the represented by a top side of this corresponds to the and the bottom rectangle to the ADU. The ADU

- Receive data side via gate from the display *dinp*.
- Output its from the gate *dout*. display status display side via
- Receive data abstraction *ainp*. from the side from gate
- Output non-display data to other interactors via gate *aout.*

Interactions on these gates modify the local state of the interactor consisting in the *abstraction* and the *display*. The ADU applies the following operations on these state components:

- The operation *input* computes the abstraction by interpreting data which is input from the display side of the interactor, with respect to the current display and the current abstraction of the interactor.
- The operation *echo* computes the display by interpreting data which is input from the display side of the interactor, with respect to the current display and the current abstraction of the interactor.
- The operation *result* interprets the current abstraction and produces data which is communicated to the application or to other interactors. There may be various *result* operations, in which case each value is offered on a different *aout* gate. In some cases this operation can be the identity operation when the value of the abstraction is sent to other interactors as it is.
- The operation *render* updates the display with respect to data received from the abstraction side. It allows the application and other interactors to modify the graphical appearance of the interactor or to visualise, through the interactor, data they send to it.
- The ADU applies operation *receive* to update the abstraction with respect to data received from the abstraction side.

The asymmetry in the handling of the abstraction and the display reflects their different roles in the ADC model. While different 'clients' of the interactor may require a different interpretation of its abstraction, computed through operation result, this is not the case for the display. As will be shown when the ADU specification is discussed and as is indicated in figure 4.1, the ADU outputs its display state without further interpretation/transformation.

The operations listed above are specified in an ACT-ONE data type *ad* with which the interactor is associated. Its signature is as follows:

```
type ad is
sorts
    abs, disp, dInpData, aInpData, aOutData
opns
    input:    dInpData, disp,abs    ->    abs
    echo:     dInpData, disp, abs   ->    disp
    render:   disp, aInpData        ->    disp
    receive:  abs, aInpData         ->    abs
    result:   abs                   ->    aOutData
endtype
```

The sorts *abs* and *disp* describe the state parameters, *dInpData* describes input arriving at the display side (gate *dinp*), *aInpData* describes input arriving at the abstraction side (gate *ainp*), and *aOutData* describes data which is sent to the 'clients' of the interactor. An abstract data type associated with a particular interactor will have custom identifiers for its sorts and operations. The signature will be amended with equations that make the specification a more meaningful specification of the interactor. In later examples it will be shown how the data type *ad* is defined for some example interactor components. Such a specification may extend domain specific data types that model the semantics of the problem domain.

The distinction between the display side and the abstraction side of an interactor is a conceptual aid for specifying an ADC interactor. When both the display state and the abstraction state are used to interpret received input, this is considered to be arriving at the display side. This corresponds to user input that the interactor needs to de-reference with respect to the current display. In practice, this is the only difference between input arriving to one side or the other. Choosing which side an interactor gate belongs to does not depend on what it connects to, but rather, on how the information received is meant to be interpreted. This distinction of the sides of the interactor is slightly different but not necessarily inconsistent to the distinction of a user and an application side used by the Pisa model, which was discussed in chapter 3. Indeed, when considering the whole of the interface as a single interactor the two views coincide.

This section has presented some syntax, i.e. the signature of the data type *ad*. Clearly, it is the structure of this signature that matters and not the names of the sorts or the operations. In reality, what this signature defines for all ADC interactors is a classification of the data it manages and the operations upon the data. It is not necessary to specify an operation called *input*, *echo*, etc., in all data types associated with the interactors. However, there is a commitment that all operations will have the purpose

and syntax of the operations defined here. This defines a consistent specification style for the data types associated with all ADC interactors.

4.4 The Abstraction Display Unit

The main purpose of the ADU is to relate action occurrences to operations of the data type ad. The ADU is a process which has some local parameters describing its state. The process definition for the ADU is quite simple. Its behaviour is a choice between action prefix expressions. There is one such expression for each gate of the ADU. An action specification prefixes the recursive instantiation of ADU with the designated operations applied to its state parameters.

```
process ADU[dinp, dout, ainp, aout](a:abs, dc,ds:disp):noexit:=
    dinp?x:dInpData; ADU[...](input(x,ds,a), echo(x,ds,a), ds) []
    dout!dc;         ADU[...](a, dc, dc) []
    ainp?x:aInpData; ADU[...](receive(a,x), render(dc,x), ds) []
    aout!result(a);          ADU[...](a, dc, ds)
endproc
```

There are two local variables holding values of type *disp.* The parameter *ds* holds the current state of the display, i.e. the last value output on the display from gate *dout*. The parameter *dc* is the computed display which the interactor will output on the next interaction on gate *dout*. The ADU does not necessarily output its most recently calculated display, so parameter *ds* is necessary to maintain a separate record of the value of the display that is used to interpret all arriving user input.

An interaction on gate *dinp* will cause the process to instantiate itself recursively, changing its local state parameters. The abstraction is set to the value produced by operation *input* and the computed display is set to the value produced by *echo*. An interaction on *dout* will output the computed display status and the process is called recursively with the current display set to the value of the computed display. An input from the abstraction side, on gate *ainp,* will set the value of the abstraction status to the value computed by operation *receive* and the computed display to the value computed by *render*. An output on the abstraction side, on gate *aout,* of a value computed by the interpretation operation *result* will have no effect on the state parameters of the ADU.

The role of the two display variables requires some explanation. Consider for example two consecutive input events, on gate *dinp,* that are not separated by an output event, on gate *dout,* between them. The second input is interpreted with respect to the same display state as the first. This is the value of the local state component *ds*. Suppose the two *dinp* events are followed by an output event *dout.* This causes the recursive instantiation of the ADU where the formal parameter *ds* is set to the last value of the *dc* parameter.

It is worth noting that the gates of the ADU are typed according to their purpose. In figure 4.2, it can be seen that on the display side input gates may receive data of sort *dInpData* and output gates will offer data of sort *disp*. On the application side input

gates may receive data of sort *aInpData* and output gates may offer data of sort *aOutData*.

The description so far shows that the ADU implements a mapping from the gates of ADU to the operations of data type *ad*. This mapping is standard for all ADC interactors. It is not the only plausible mapping that can be used for the same purpose and it is not argued that it is necessarily the best. The mapping was shaped on the basis of its intuitive appeal and it was refined through the practical application of the model. Of more significance is that this mapping is standardised for all interactors. Provided it does not unduly restrict the expressiveness and malleability of the representation scheme, then the specifier-user of the ADC model is relieved of the task of relating the dynamic component of the specification to the data typing component. The ADU specification follows mechanically from a definition of the data type *ad*, and a characterisation of what is the role of each gate of the interactor, e.g. input on the display side, output from the abstraction side, etc.

4.5 The Constraints Component

As mentioned above the ADU does not specify a temporal ordering on its behaviour. This is the purpose of another process, called the Constraints Component (CC). The CC constrains the behaviour of the ADU to exhibit a certain pattern by synchronising on all its gates. By definition, the gate set of the CC includes all the gates of the ADU, plus some more which may be useful in describing dialogue constraints, e.g. synchronisation with other interactors, triggers, etc. For the sake of the presentation assume there is one such gate *d* that stands for a set of gates $G_{dialogue}$ of the same purpose. These gates belong to the alphabet of process CC but not to the alphabet of process ADU. Dialogue (or control) gates are assigned to the controller side of the interactor, represented in the illustrations by the sides of representation of the interactor. The composition of the ADU with the simple CC component is shown in figure 4.2. A simple model of the ADC interactor can be obtained as the synchronous composition of the ADU and the CC over all the gates of the ADU, which from now on will be called *the set of input and output gates* $G_{io.}$

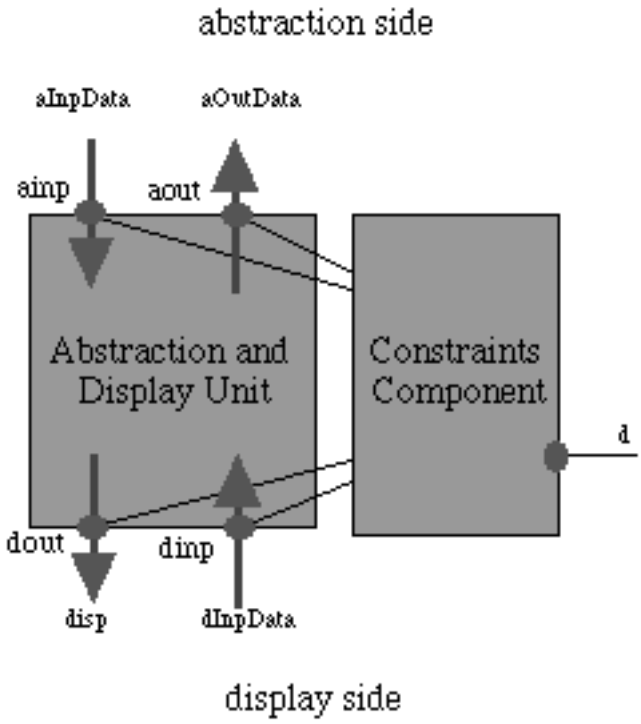


Figure 4.2 Composition of the ADU and the constraints component CC.

```
process ADC[d, dinp, dout, ainp, aout] (A:abs, D:disp):noexit :=
    ADU[dinp, dout, ainp, aout](A, D, D)
            |[dinp, dout, ainp, aout]|
    CC[d, dinp, dout, ainp, aout]
endproc
```

The CC is a simplified form of the CU which is defined in section 4.7. There the CC is defined as a component of the CU. The CU will be defined as a parameterised structure that provides the SSRRA behaviours as well. Below is an example of a CC that forces an echo after each input event and sets no other constraint:

```
process CC[d, dinp, dout, ainp, aout] : noexit :=
      ainp?X:aInpData;    dout?X:disp;    CC[d, dinp, dout, ainp, aout] []
      aout?X:aOutData;                    CC[d, dinp, dout, ainp, aout] []
      dout?X:disp;                        CC[d, dinp, dout, ainp, aout] []
      dinp?X:dInpData;    dout?X:disp;    CC[d, dinp, dout, ainp, aout] []
      d;                                  CC[d, dinp, dout, ainp, aout]
endproc
```

Actions on the gates of G_{io} = {dinp, dout, ainp, aout} are typed, in accordance to the ADU. However, the CC does not apply any selection on the values offered and does not record this information in some local state parameter. Other than the typing information on its gates, which is necessary for the synchronisation with the ADU, the CC ignores the data values associated with events.

As a second example, consider that there is an input trigger (*it*) and an output trigger event (*ot*). Their purpose is equivalent to those of the Pisa interactor model discussed in

section 3.6. The input trigger should cause an event on gate *aout* and the output trigger event should cause an event on *dout*.

```
process CC[it, ot, dinp, dout, ainp, aout]:noexit :=
        ainp?X:aInpData;          CC[it, ot, dinp, dout, ainp, aout] []
        it;   aout?X:aOutData;    CC[it, ot, dinp, dout, ainp, aout] []
        ot;   dout?X:disp;        CC[it, ot, dinp, dout, ainp, aout] []
        dinp?X:dInpData;          CC[it, ot, dinp, dout, ainp, aout]
endproc
```

This example demonstrates the versatility of using the constraint oriented style to specify the interactor behaviour. Here, the black box version of the Pisa interactor model, specified in basic LOTOS in [54], has been specified in the CC component. The synchronisation of CC with the ADU specifies the interactor in full LOTOS. Any alternative behaviour can be specified in the CC. Further constraints can be added incrementally by parallel composition. In contrast, consider the white box version of the Pisa interactor model [150], which is specified as a composition of resources. It may not always be possible to specify and modify its behaviour by applying constraints externally. Instead it may be necessary to modify the specification of the resources comprising the interactor.

The example above, illustrates also how the use of the model is generalised. The gate identifiers used in the LOTOS definitions are place holders for gate sets of similar purpose. In this case the gate *d* in the definition of the CC component corresponds to a gate set $G_{dialogue}$= {it, ot}.

4.6 A simple example: The specification of a scroll bar

The specification of interactor *scr* modelling a simple scroll bar is presented, starting from its corresponding data type *scr_ad*. The specification of *scr_ad* combines the specifications of *scrBar* and *BoundedVal*. Data type *scrBar* describes the visible representation of a scroll bar. *BoundedVal* represents the abstract notion of a bounded value which the scroll bar manipulates. An operation *input* pertains to the mapping of a point to an integer value linking the two data types. Only the syntax for this operation is defined. There are two operations *render*. Respectively, they describe the resizing of the scroll bar and the visualisation of a bound value. Operation *receive* substitutes the abstraction with a new bound value input to the interactor. Operation *result* returns the integer value of the abstraction.

```
type scr_ad is  scrBar, boundedVal
opns
    input:    pnt, scrollbar, boundValue ->      boundValue
    echo:     pnt, scrollbar, boundValue ->      scrollbar
    render:   scrollbar, boundValue              ->      scrollbar
    render:   scrollbar, rct                     ->      scrollbar
    receive: boundValue, boundValue              ->      boundValue
    receive: boundValue, rct                     ->      boundValue
    result:   boundValue                         ->      Int
```

```
    initBV:                                  ->      boundValue
    initSB:                                  ->      scrollbar
eqns
forall r:rct, p:pnt, sb: scrollbar,  bv1,bv2: boundValue
ofsort boundValue
    receive(bv1, bv2) = bv2;
    receive(bv1, r)=bv1;
ofsort scrollbar
    echo(p,sb,bv1) = changePnt(sb, p);
    render(sb, r) = changeRect(sb, r);
    render(sb, input(p,sb, bv1))=changePnt(sb,p);
ofsort Int
    result(receive(bv1, bv2))=val(bv2);
endtype
```

The data types *BoundedVal* and *scrBar* do not follow the syntactic conventions defined by the signature of type *ad*. For the sake of a complete presentation, their specification is included below. *BoundedVal* supports three inquiry operations that return integer values for the upper and lower bounds and the value that ranges between them. Three transformer operations are defined to set this value.

```
type BoundedVal is Integer
sorts
    boundValue
opns
    lo, val, hi:        boundValue        ->      Int
    setVal:             boundValue, Int   ->      boundValue
    incr, decr:         boundValue        ->      boundValue
eqns
    forall b:boundValue, n: Int
ofsort Int
    val(setVal(b, n)) = n;
    val(incr(setVal(b,n))) = s(n);
    lo(setVal(b,n)) = lo(b);
    hi(setVal(b,n))= hi(b);
ofsort bool
    lo(b) le val(b) = true;
    val(b) le hi(b) = true;
ofsort boundValue
    incr(b) = s(val(b));
    decr(b) = p(val(b));
endtype
```

A simple and abstract model of a scroll bar is characterised by two graphical entities: a rectangle and a point. The data type *Graphics* which models such basic graphical entities is extended in the definition of *scrBar*, but it is not necessary to describe it below. The operations on a scroll bar may change or return the value of the rectangle and the point.

```
type scrBar is Graphics
sorts
    scrollbar
opns
    mkscrollBar:        rct, pnt          ->      scrollbar
```

```
    changePnt :     scrollbar, pnt    ->      scrollbar
    changeRect:     scrollbar, rct    ->      scrollbar
    rect:           scrollbar->       rct
    point:          scrollbar->       pnt
eqns
    forall p:pnt, r:rct, sb:scrollbar
ofsort rct
    rect(mkscrollBar(r,p)) = r;
    rect(changeRect(sb,r)) = r;
    rect(changePnt(sb,p))=rect(sb);
ofsort pnt
    point(mkscrollBar(r,p)) = p;
    point(changeRect(sb, r)) = point(sb);
    point(changePnt(sb,p))=p;
endtype
```

The data types *scrBar* and *BoundedVal* are not sufficiently complete and they lack detail. However, this is not necessary in order to exemplify the specification of the scrollbar interactor and would introduce unnecessary clutter. The specification of the entities displayed, like *scrBar* above, may use a standard model of the display at a uniform level of abstraction for all interactors. The specification data type for the abstraction like *BoundedVal* in the last example may be used to specify the domain semantics. Such specifications could be refined to increasing levels of detail without changing the interactor specification. These issues are discussed in a later chapter. This brief discussion aims simply to draw the attention of the reader to the fact that the interactor specifications do not provide a complete model of the system nor a model of the display. This is consistent with the scope of the model, which is only the user interface system portion of the interactive system (see the Arch reference model of chapter 2). The interactor-specific data types may link to such specifications if a complete model of the interactive system is desired.

The ADU for the scrollbar *scrADU* is obtained from the general process definition ADU, by simple substitution of the gate identifiers and the operations from the data type *scr_ad*.

```
process scrADU[dinp, dout, ainp, aout](a: boundValue, dc, ds: scrollbar) : noexit :=
    aout!result(a);              scrADU[dinp, dout, ainp, aout](a, dc, ds) []
    dout!dc;          scrADU[dinp, dout, ainp, aout](a, dc, dc) []
    ainp?x:rct;                  scrADU[dinp, dout, ainp, aout](receive(a,x),render(dc,x), ds) []
    ainp?x:boundValue;           scrADU[dinp, dout, ainp, aout](receive(a,x), render(dc,x), ds)[]
    dinp?x:pnt;                  scrADU[dinp, dout, ainp, aout](input(x,ds,a), echo(x,ds,a), ds)
endproc
```

The interactor *scr* describing the scrollbar can be obtained with the synchronous composition with a process *scrCC* which, in this case, enforces immediate feedback for all input from both the display and the abstraction sides:

```
process scr[dinp, dout, ainp, aout] (A:boundValue, D:scrollbar):noexit :=
    scrADU[dinp, dout, ainp, aout](A, D, D)
            |[dinp, dout, ainp, aout]|
    scrCC[dinp, dout, ainp, aout]
endproc
```

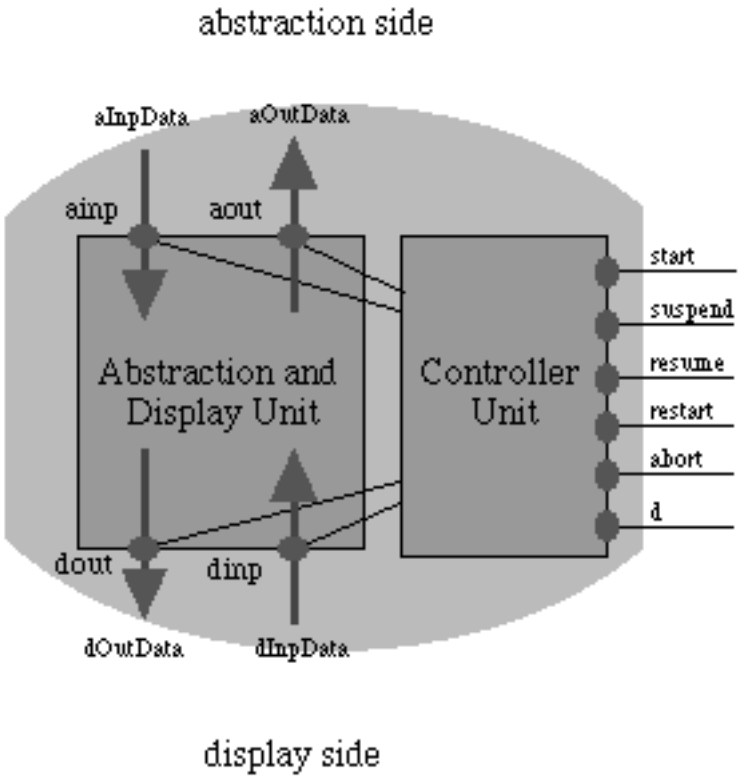


Figure 4.3. The internal structure of the ADC interactor.

```
process scrCC[dinp, dout, ainp, aout] : noexit :=
    ainp?X:rct;          dout?X:scrollbar;  scrCC[dinp, dout, ainp, aout] []
    ainp?X:boundValue; dout?X:scrollbar;          scrCC[dinp, dout, ainp, aout] []
    aout?X:Int;                                   scrCC[dinp, dout, ainp, aout] []
    dout?X:scrollbar;                       scrCC[dinp, dout, ainp, aout] []
    dinp?X:pnt;          dout?X:scrollbar;  scrCC[dinp, dout, ainp, aout]
endproc
```

4.7 The Controller Unit

A slightly more complicated version of the ADC interactor model is introduced in this section. The ADC interactor is formed by the parallel composition of two processes, the ADU and the CU as follows:

```
process ADC[start, suspend, resume, restart, abort, d, dinp, dout, ainp, aout] (A:abs, D:disp):noexit :=
    ADU[dinp, dout, ainp, aout](A, D, D)
            |[dinp, dout, ainp, aout]|
    CU[start, suspend, resume, restart, abort, d, dinp, dout, ainp, aout]
endproc
```

The CU constrains the temporal behaviour of the ADU just like the CC in the previous section. It also specifies behaviours found commonly across user interface objects. These behaviours are effected by action occurrences on corresponding gates.

Collectively they are referred to as SSRRA behaviours (respectively gates) and they are defined as follows:

- The operation of the interactor starts with an action *start*.
- The operation is suspended with an action *suspend*.
- The operation is resumed with an action *resume.*
- At any point during interaction the operation can be restarted with an action *restart* after which the temporal behaviour of the interactor is the same that follows a *start* action.
- At any point the operation of the interactor can be stopped with an *abort* action.

A comment is warranted before the formal definition of the CU. There is a limit to the use of parameterisation and it is questionable whether the interactor model benefits from the introduction of these new features. Supporting such behaviours should be beneficial to the specifier who uses the model: they are very common for user interface components and their specification is not trivial. The set of SSRRA behaviours that are supported in the definition of the CU was chosen after the GARNET user interface management system [136, 137]. All GARNET interactors support the same parameterised state machine whose behaviour is very similar to the SSRRA behaviours. The GARNET state machine provides a feasible and viable proposal for the parameterisation of user interface behaviours. Practical experience with using the ADC model may eventually corroborate this choice or lead to changes to it. In fact, the CU definition below already incorporates modifications to the original model of [123], that stem from experience of the case study reported in chapter 5.

The general structure of the CU is shown in the process definitions below.

```
process CU[start, suspend, resume, restart, abort, d, dinp, dout, ainp, aout] : noexit :=
    start; RUN [suspend, resume, restart, abort, d, dinp, dout, ainp, aout]
where

process RUN[suspend, resume, restart, abort, d, dinp, dout, ainp, aout] : noexit :=
    (       CC[d, dinp, dout, ainp, aout] |[d, dinp, dout, ainp, aout]|
            SU_RE[suspend, resume, d, dinp, dout, ainp, aout])
[>  INTERRUPT[suspend, resume, restart, abort, d, dinp, dout, ainp, aout]
endproc

process CC[d, dinp, dout, ainp, aout] : noexit :=
    ainp?X:aInpData;            CC[d, dinp, dout, ainp, aout] []
    aout?X:aOutData;            CC[d, dinp, dout, ainp, aout] []
    dout?X:disp;                CC[d, dinp, dout, ainp, aout] []
    dinp?X:dInpData; CC[d, dinp, dout, ainp, aout] []
    d;                          CC[d, dinp, dout, ainp, aout]
endproc
process SU_RE[suspend, resume, dinp, d, dout, ainp, aout]: noexit :=
    ANYORDER[d, dinp, dout, ainp, aout]
[>  suspend; resume; SU_RE[suspend, resume, d, dinp, dout, ainp, aout]
endproc
```

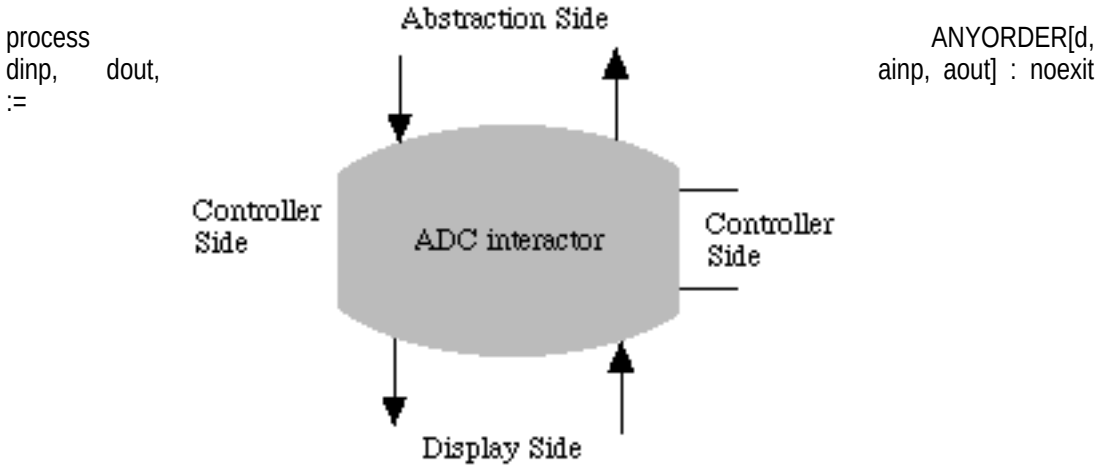


Figure 4.4. Convention for the representation of ADC interactors.

```
    ainp?X:aInpData;          ANYORDER[d, dinp, dout, ainp, aout] []
    aout?X:aOutData;          ANYORDER[d, dinp, dout, ainp, aout] []
    dout?X:disp;              ANYORDER[d, dinp, dout, ainp, aout] []
    dinp?X:dInpData;          ANYORDER[d, dinp, dout, ainp, aout] []
    d;                        ANYORDER[d, dinp, dout, ainp, aout]
endproc

process INTERRUPT[suspend, resume, restart, abort, dinp, d, dout, ainp, aout]: noexit :=
    restart; RUN[suspend, resume, restart, abort, d, dinp, dout, ainp, aout]
[]  abort; stop
endproc
endproc
```

Process CC above is rather contrived. It induces a clockwise (on figure 4.3) ordering of events on the gates of the ADU. This CC is included simply as a place holder for more meaningful specifications. Process SU_RE describes the suspension behaviour of the interactor: it halts with the *suspend* action and resumes where it was with a *resume*; the rest of the time it allows any order of action occurrences on the gates in $G_{dialogue}$ G_{io}. A *restart* action has the effect of restarting the constraints process, so that the local dialogue returns to its initial state, i.e. the one that follows a *start* action. An action *abort* terminates the operation of the interactor and the process exits.

Figure 4.3 illustrates the interactor as a barrel shape. The top and bottom arches of the barrel-shape are dedicated to the abstraction and display side of the interactor respectively. The vertical sides are called the controller sides of the interactor, and they are dedicated to gates of the CU which are not also gates of the ADU. Gates of the interactor are marked by incoming arrows for input gates, outward arrows for output gates, and lines for simple synchronisation gates. Figure 4.3 shows substantial detail that is omitted when the interactor is used as a unit for building higher level structures. The two components of the ADC, the ADU and the CU, and their connections do not need to be shown and the interactor is more economically represented as in figure 4.4.

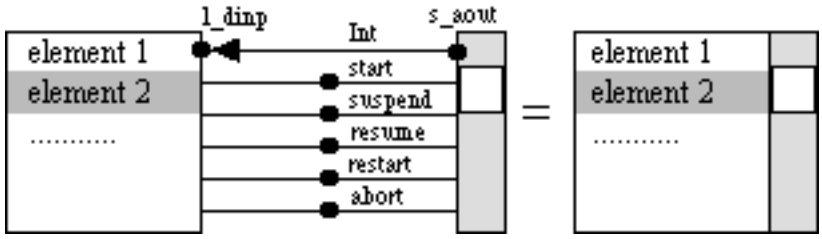


Figure 4.5. A scrollable list as a composition of a list-window and a scrollbar.

4.8 The scrollable list example: composition of two interactors

The specification of an idealised and simplified scrollable list is presented. It does not describe an interactor specific to any particular interface system. The list may contain various items, e.g. icons, strings, etc. This is not specified in the example, although figure 4.5 illustrates a list of text fields. The list is observed through a window whose contents depend on the window size and the position of the window relative to the displayed list. For example, the position may be defined by the index of the first list element displayed. The user may scroll up and down the list by using a scrollbar.

The scrollable list is defined as the composition of a scrollbar interactor, called *scr*, and an interactor modelling the interactive list, named interactor *lst.* Interactions start, suspend, resume, restart, and abort together, since they support the same interaction task, so the interactors synchronise on the SSRRA gates of the CU. The specification of interactor *scr* has been described in section 4.6. Here, it is only necessary to discuss interactor *lst* and their combined operation. The only change introduced with this example is that the constraints component *s_CC* of interactor *scr* is embedded within a controller unit *s_CU* as discussed in section 4.7.

The scrollbar receives input from the user as a cursor position and interprets this input

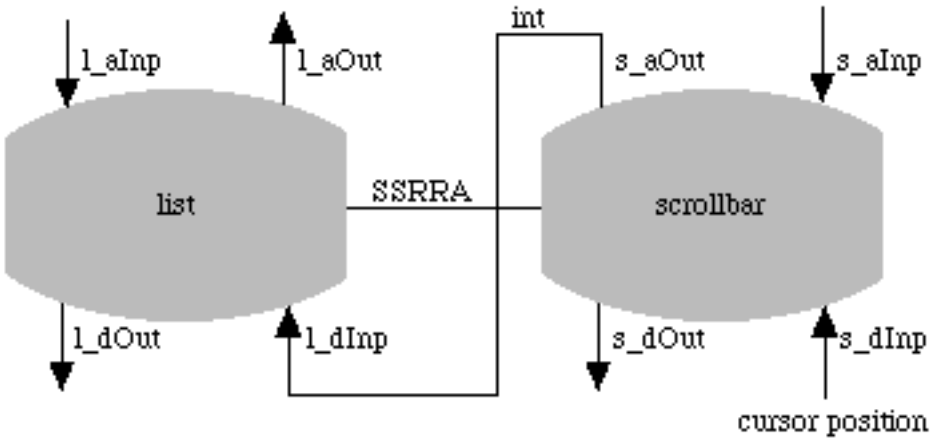


Figure 4.6. The composition of the two interactors.

producing an integer value. The integer value is passed via the gate *s_aout* of the scrollbar interactor as an input to the gate *l_dinp* of the list. The list interprets the integer value to produce a new starting position for the window, thus achieving the effect of scrolling. The configuration of the composed interactors is illustrated in figure 4.6. The lines and arrows connecting the two interactors represent their synchronisation on the corresponding gates.

The data type *ad* for the list interactor is *ls_ad* defined as follows:

```
type ls_ad is lstElements, scrList
opns
    input:    pnt, scrLst, lstel  ->    lstel
    echo:     pnt, scrLst, lstel  ->    scrLst
    input:    Int, scrLst, lstel  ->    lstel
    echo:     Int, scrLst, lstel  ->    scrLst
    render:   scrLst, lstel       ->    scrLst
    receive:  lstel, lstel->      lstel
    result:   lstel               ->    el
    initLst:                      ->    lstel
    initScrLst:                   ->    scrLst
eqns
forall m: Int, p:pnt, s:scrLst, w,wold,wnew:lstel
ofsort  lstel
    input(p, s, w) = sel(w,pick(s,p));
    input(m, s, w) = setstart(w,m);
    receive(wold, wnew) = wnew;
ofsort scrLst
    echo(p, s, w) = changeLne(s, pick(s,p));
    render(render(s, wold), wnew) = render(s,wnew);
ofsort el
    result(w)=which(w);
endtype
```

The data type *ls_ad* combines the type *lstElements*, which models a list of elements, and *scrList*, which models the window display for the list. Its signature derives by substitution from the general definition of *ad*. The sort of the abstraction parameter is *lstel* and the sort of the display parameters of the interactor is *scrLst*. *lstElements* defines an enquiry operator *which(lstel)* that returns the selected element of the list. *scrList* is associated with operation *pick(scrLst,pnt)* which returns an index of the displayed window (i.e. a line or icon number) given a cursor position *pnt.* An element selection for *lstElements* can be set by *sel(lstel, Int)*. Finally, the position of the window with respect to the list, is set by operation *setstart(lstel, Int)* which uses the data sent from the scroll bar.

```
type lstElements is Integer
sorts
    lstel, el
opns
    sel:          lstel, Int      ->    lstel
    setstart:  lstel, Int         ->    lstel
    which:        lstel           ->    el
    wnstart:   lstel              ->    Int
eqns
```

```
forall W:lstel, N,M:Int
ofsort Int
    wnstart(setstart(W, N)) = N;
    wnstart(sel(W, N)) = wnstart(W);
ofsort lstel
    sel(sel(W,N),M)=sel(W, M);
    sel(setstart(W, N), M)=sel(W,M);
ofsort el
    which(setstart(W, M)) = which(W);
endtype

type scrList is Integer, Graphics
sorts
    scrLst
opns
    mkscrLst:       rct, Int        ->      scrLst
    changeLne:      scrLst, Int     ->      scrLst
    changeRect:     scrLst, rct     ->      scrLst
    pick:           scrLst, pnt     ->      Int
    rect:           scrLst          ->      rct
    line:           scrLst          ->      Int
eqns
    forall l:Int, r:rct, sl:scrLst, p:pnt
ofsort rct
    rect(mkscrLst(r,l)) = r;
    rect(changeRect(sl,r)) = r;
    rect(changeLne(sl,l))=rect(sl);
ofsort Int
    line(mkscrLst(r,l)) = l;
    line(changeRect(sl, r)) = line(sl);
    line(changeLne(sl,l))=l;
    pick(changeLne(sl, l), p)=pick(sl,p);
endtype
```

The ADU for the list is *l_adu*. Once more its definition follows mechanically from the general form of ADU.

```
process l_adu[dinp, dout, ainp, aout](a: lstel, dc, ds: scrLst) : noexit :=
    aout!result(a);     l_adu[dinp, dout, ainp, aout](a, dc, ds) []
    dout!dc;  l_adu[dinp, dout, ainp, aout](a, dc, dc) []
    ainp?x:lstel;       l_adu[dinp, dout, ainp, aout](receive(a,x), render(dc,x), ds) []
    dinp?x:pnt;         l_adu[dinp, dout, ainp, aout](input(x,ds,a), echo(x,ds,a), ds)[]
    dinp?x:Int;         l_adu[dinp, dout, ainp, aout](input(x,ds,a), echo(x,ds,a), ds)
endproc
```

The controller is the same as in the general case; only CC needs to be modified with the sort identifiers of data type *ls_ad*. The constraint described is that the interactor will immediately inform the user of data it receives from its application side.

```
process CC[dinp, dout, ainp, aout] : noexit :=
    ainp?X:lstel;       dout?X:scrLst;    CC[dinp, dout, ainp, aout] []
    aout?X:el;                            CC[dinp, dout, ainp, aout] []
    dout?X:scrLst;                        CC[dinp, dout, ainp, aout] []
    dinp?X:int;                           CC[dinp, dout, ainp, aout]
endproc
```

The composition of the two interactors, shown schematically in figure 4.6, is written as follows. Note the renaming of gate *l_dinp* of the list interactor to *s_aout* to support the synchronisation with the scrollbar.

```
scr[s_dinp, s_dout, s_ainp, s_aout, start, suspend, resume, restart, abort](initBV, initSB)
        |[start, suspend, resume, restart, abort, s_aout]|
lst[s_aout, l_dout, l_ainp, l_aout, start, suspend, resume, restart, abort](initLst,initScrLst)
```

4.9 Some first comments on the ADC interactor model

The ADC interactor model can be seen, rather pessimistically, as little more than syntax. The ADC interactor is a parameterised template that is instantiated to produce interactor specifications. However, syntax is very important. It is true that interactors can be modelled in many specification languages and programmed in any reasonable programming language. It is important though that the constructs of the specification language are meaningful, i.e. they correspond to entities that are meaningful in the context of user interface design and its semantic interpretation captures salient properties of user interface objects.

The ADC model extends the syntax of LOTOS with syntactic structures that describe interactors. The alternative to this approach is to use the language as it is or to extend its semantics. Vissers et al. [180] argue that “it is impractical, if not infeasible, to develop and use specification languages that contain primitive constructs for each potential architectural requirement”. Instead they propose the development of an adequate set of generic language elements and construction rules, that allow the faithful expression of architectural requirements as a composition of language elements. This statement describes accurately the role of the ADC interactor model in the specification of user interface software. Rather than using the language as it is, Gaudel [73] and Vissers et al. [180] argue for the need to structure specifications to master their size and complexity. They suggest that the required structure is particular to the field of application of the formal method. A similar rationale prompts interface developers to use software architectures (cf. [37] for a discussion regarding the purpose and requirements from software architectures for the user interface). The ADC interactor model provides a method to structure LOTOS specification which reflects the nature of user interface software.

Extending the semantics of a specification language may have some advantages. For example extensions to LOTOS have been proposed that support suspension and resumption of processes [101]. This would significantly simplify the specification of the CU. An advantage of the approach presented here is that it is consistent with existing tool support [30, 68 and 119] with the current international standard [100].

Finally, the motivation for building the interactor model is not that the semantics of the language are inadequate, but rather the need to model the semantics of interactive objects in a generalised form. In this sense the ADC model is more than syntax. It is an interpretation in LOTOS of a conceptual framework for modelling interactive systems

that postulates what an interactor is, what its components are, and how interactors can be combined to model interface software. The ADC model may be applied to any level of abstraction. An elementary interactive unit may be modelled as an ADC interactor and the same applies to the whole of the user interface. It is not suggested that the modelling framework was developed independently of its interpretation as a LOTOS specification template. The two were developed in tandem and are not independent of each other.

The validity of the ADC model as an architectural abstraction is corroborated by a comparison with the informal software architectures discussed in chapter 2. Further, as will be discussed extensively in chapter 7, it has been designed to model salient properties of interface software, previously expressed in terms of abstract models of interaction, discussed in chapter 3. However, the validity of the ADC model may only be hypothesised at this stage. Comparisons with related research and its application in case studies may be used to refute it or to progressively corroborate its validity. In chapter 5 a case study undertaken with this intention is presented.

Chapter 2 discussed user interface software architectures. It identified a trend towards object-based architectures, where objects are triplets of lower level entities. These lower level entities are similar, in purpose, to the components of the ADC interactor (abstraction, display and controller). Object based architectures vary with respect to how tightly these entities are coupled, how they are represented and how objects themselves are composed to build the interface. The same issues arise in the definition of the ADC model. It is interesting to step back and compare the ADC as a conceptual framework with the PAC model [39], which was discussed in chapter 2. PAC is mentioned because its components are reminiscent in purpose and naming to those of ADC. PAC supports a distinct representation for the control component, but it does not prescribe how the control information should be represented. The similarity does not stop there. PAC is recursive in nature, in that the whole interface and its elementary components are modelled by a PAC object. Similarly the user interface may be described as a single ADC interactor or as a composition of ADC interactors. There is an important difference between the ADC and PAC interactors: the PAC controller handles all communication between the presentation and abstraction components and translates data between the two representations, maintaining their correspondence. Such communication is ‘hidden’ from the ADC controller which simply imposes external ‘dialogue’ constraints on their operation.

PAC is an informal architecture, although it has been the subject of some formalisation attempts. For example Abowd [1] uses his Agent model to formalise the notion of correspondence between abstraction and display that the controller component of PAC maintains. Also [1] proposes a formal model of MVC [113] which is compared to that of PAC. A different object based specification of PAC and MVC is reported in [98]. It is debatable whether [1] and [98] are valid formalisations of PAC and MVC. The doubts arise precisely because these conceptual models have not been introduced with a formal specification, but rather, their definitions rely on intuitions and heuristic guidelines for structuring user interface software. This suggests that formal interactor models can help formulate in an unambiguous form user interface software architectures.

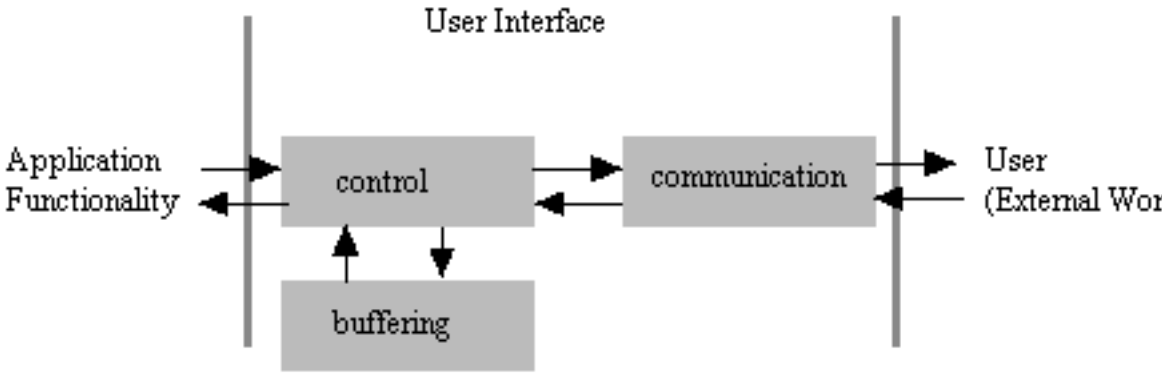


Figure 4.7. The user interface model of Kovacevic (adapted from [112]).

The idea of structuring the whole of the interface as a single ADC interactor has a direct analogue in software architectures as well. For example, Kovacevic [112] proposes a 'macroscopic' model for a user interface architecture (figure 4.7). He distinguishes a communication component which is a passive transducer of information, converting it from an external user oriented representation to an internal application oriented one and vice versa. The temporal structure of the interaction is defined in a controller component with the aid of a buffering component. The similarity with the components of the ADC model is obvious. The most outstanding difference is that the model does not support an object based architecture, i.e. the interface is not seen as a composition of many instances of this model but rather as a monolithic construction that follows the pattern suggested. When the whole interface is considered as a single ADC interactor, then the ADU and the CU correspond directly to the communication and controller unit of the model of [112]. More generally, layered architectures like the Arch reference model [14] and its predecessor the Seeheim model [77] may be formalised as a single ADC interactor.

Compared to the Pisa formal interactor model, the ADC interactor introduces a complete separation of roles for the ADU and the CU components. This modular description of control information aims to make ADC more usable as a design oriented representation providing two orthogonal dimensions for the representation scheme. The temporal ordering constraints on the behaviour of the interactor are represented as a set of constraints applied to its externally observable behaviour. This contrasts the specification of the intensional specification of the Pisa interactor (see figure 3.8), which was described as the composition of lower level entities. The difference is that between a resource oriented specification and a constraint oriented specification [179]. The resulting advantage is that the ADC model allows for the easy inspection and customisation of the interactor by applying constraints on its observable behaviour. What enables this departure is that the proposed model does not formalise the components of the GKS reference model and can therefore be made more flexible.

4.9.1 Modelling interfaces as composition graphs

Interfaces are modelled as compositions of interactors. The composition of two or more interactors has two facets:

- The composition of their effect on the data transmitted.
- The composition of their controller specifications.

Each interactor applies operations to the data passing through it, as specified in the ADU component. Directing the output of one interactor to the other, by their synchronous composition, will have the effect of composing the operations they apply. For example, the combination of the list interactor and the scroll bar interactor applies an operation *input(result(input(...)),scrLst,lstel)* on the mouse input, where the inner most input operation belongs to the data type *scr_ad* and the outermost input belongs to *ls_ad*.

When two interactors are connected over a pair of gates, it is required that the domain of the data that is output by one is a subset of the domain that can be received by the second. This is a well-formedness constraint for the composition of interactors which will be discussed more in chapter 6. Provided this condition holds, the temporal ordering of a composition of interactors is sufficiently described by the composition of their controller units. This simple observation is a compositionality property, specific to the ADC interactor model. It results from the orthogonality in the description of its components and it is supported in the formal specification by the multi-way synchronisation of LOTOS.

4.9.2 Compositionality of the ADC model

The compositionality suggested above means that the composition of two ADC interactors is also, or rather it can be formed as, an ADC interactor. For example, the composition of the previous section is equivalent to an ADC which has the form:

```
adu[s_dinp, s_dout, s_ainp, s_aout, l_dout, l_ainp,l_aout](initBV, initSB, initLst, initScrLst)
    |[s_dinp,s_dout, s_ainp, s_aout, l_dout, l_ainp, l_aout]|
cu[start, suspend, resume, restart, abort, s_dinp, s_dout, s_ainp, s_aout, l_dout, l_ainp, l_aout]
```

where the ADU for this interactor is defined as

```
process adu[...](BV:boundValue, SB:scrollbar, L:lstel, SL:scrLst):noexit:=
s_adu[s_dinp, s_dout, s_ainp, s_aout](BV, SB, SB)
    |[s_aout]|
l_adu[s_aout, l_dout, l_ainp, l_aout](L, SL, SL)
endproc
```

and the controller is defined as

```
process cu[...]:noexit:=
s_cu[start, suspend, resume, restart, abort, s_dinp, s_dout, s_ainp, s_aout]
    |[start, suspend, resume, restart, abort, s_aout]|
l_cu[start, suspend, resume, restart, abort, s_aout, l_dout, l_ainp, l_aout]
endproc
```

This observation is described and proven rigorously in the chapter 6. Capitalising on this property, transformations of ADC specifications are defined formally.

4.10 Summary

This chapter has presented the formal specification of the ADC interactor model. The ADC model was introduced gradually starting from a simpler model which does not support parameterised control behaviours. The full ADC model supports a standard and parameterised behaviour for all interactor instances. This behaviour is similar to the state machine for the interactors of the GARNET user interface management system.

An ADC interactor transforms the data passing through it by applying operations defined in an ACT-ONE data type *ad*. The data type specification is linked with the operational description of the interactor by a process called the ADU. The ADU does not exhibit any temporal ordering in its behaviour. Its state can be described sufficiently by the value of its abstraction and the display state parameters, without reference to the history of events that caused it. On the contrary the state of the CU, which specifies the temporal structure of the observable behaviour, is adequately described by its 'dialogue state', e.g. by an equivalence class of histories of events. This orthogonality in the description of the ADC gives rise to a property of compositionality briefly exemplified with the specification of a scrollable list. Finally, this chapter included a brief comparison of the structure and the purpose of the components of the ADC model and other formal and informal architectures.

The ADC model provides a conceptual framework for modelling user interface software. Interface software may be modelled as a single unit or as a structured composition of lower level entities. Both approaches have direct analogues in the domain of user interface software. The specification may address different levels of abstraction. The level of abstraction is determined by the granularity of the actions specified and the granularity of the data managed by the interactor.

Using ADC as a specification template provides a head-start to the specification activity. The model defines the dimensions for the description of interactive objects and, by means of LOTOS temporal operators, a way of building up structured compositions of such objects. The model defines a mix of specification styles: resource oriented for the description of the ADU and constraint oriented to describe its temporal behaviour. This chapter has argued in favour of standardising the specification style in this manner. This standard style should make it easier to write specifications and should facilitate people other than their author in reading and writing specifications.

Chapter 5

A case study in the use of the ADC interactor

This chapter reports a case study in the use of the ADC interactor model. The case study concerns a reverse engineered specification of the graphical interface of Simple Player™, which is an application for playing QuickTime™ movies on the Macintosh computer. The case study is reported in full in [124]. Since that report was written several improvements have been made both to the product specification and to the ADC model itself. To a great extent these changes have resulted from the case study. Only a brief summary of the specification and the process of its creation is presented below. Taking up a methodological issue, this chapter discusses the limitations of the case study as a scientific assessment of the ADC model and the limitations of scale that such studies need to address. The discussion suggests some qualifiers regarding the conclusions that the case study may lead to. Still, it is upheld as a useful and necessary experience in using the ADC model. Overall the case study was successful in that it demonstrates the use of the model and its feasibility as a representation scheme. Most important it has prompted numerous improvements to the model which are summarised in the end of this chapter.

5.1 Motivation for the case study

The ADC model was developed through a series of elementary examples like those used in the exposition of chapter 4. These examples concerned fictional and idealised interactive components, like buttons, sliders, and menus. They have provided useful feedback in the early stages of the model's development. However, formal models and particularly those applied to the study of the human computer interface need to be tested against larger scale problems. The specification of idealised interaction styles, that do not refer to a particular implementation platform or style-guide, does not constitute a real test for the modelling technique. If the requirement for realism is relaxed, it is inevitable that the specificand will be moulded to fit the model instead of testing it. Therefore, it is necessary to test the ADC interactor model in a problem of realistic size and complexity.

A model, formal or not, is put forward with a particular application domain in mind. An example application of the model is characterised by how representative it is of the domain and by its coverage of this domain. For example, the ADC should be applied to representative examples of graphical interaction as this was its target domain of application. A credible test for the interactor model should cover as wide as possible a range of interaction styles.

A single example application is not conclusive on its own. Numerous successful examples are necessary to accrue confidence in the model. A single example application may be seen as a test of the feasibility of using the model in the target application domain and an assessment of the practicality of doing so. The conclusions drawn from such an application are only indicative of its appropriateness. In principle, the case study may lead to the rejection of the model, on the grounds that it is insufficient for the intended application domain, or that it is too cumbersome to use. In practice, the aim is to highlight limitations of the model in its current form so that they may be corrected.

The conclusions drawn from an assessment exercise depend upon how the model is used. There is a gap between the academic application of a model, particularly from the person who proposes it and its application in a realistic design project. The latter is only possible when the model has developed to a considerable level of maturity. Intermediary levels of its application may involve, e.g. its application for academic purposes by people other than the creator of the model, experimental application under controlled circumstances, the experimental application by practitioners, the use of the model in anger, etc. Unfortunately, it takes a long time before a model is sufficiently mature and has been widely enough disseminated so that its application may be assessed scientifically.

To summarise, this discussion has identified the following issues pertaining to assessing a model through example applications:

- The software modelled should be of realistic size and complexity.
- Examples should provide sufficient coverage and representativeness for the target domain.
- Numerous examples are needed to accrue confidence in the model.
- The conclusions drawn depend upon the rigour of the assessment method.

The ADC model is still at an early stage of its development, so it is not yet practical to test its use by one or more independent specifiers in a realistic design project. The required rigour for a scientific assessment of its use is not yet possible. Rather than directing the effort into numerous small scale examples, a single sizeable application of the model was undertaken. The case study is not put forward as an adequate assessment of the model. Instead it has had a ‘formative’ role in the sense that it provided the opportunity for significant changes and improvements to the model. The specification itself is a useful product of the study, necessary for disseminating the ideas built into the ADC model through a concrete example.

In spite of its modest formative role, the case study still requires an objective measure of success or failure. A reverse engineering application of the ADC model to specify an existing interface fulfils this requirement. By simulating the LOTOS specification, it is relatively straight forward to compare the behaviour predicted by the model to that exhibited by the actual software. The interface software was modelled to a high level of detail. This offers a concrete criterion for comparing the specified behaviour with the observed behaviour of the system. It can be claimed that the interaction is specified precisely and difficult issues are not brushed over. On the down side, opting for a low abstraction level is detrimental towards the magnitude of the study. More sizeable software can be modelled by adopting a higher level of abstraction.

5.2 Simple Player™: the subject of the case study

QuickTime™ is a system-software extension for the Apple Macintosh computer. It is used by application programs as a functional interface to work with media such as sound, video, and high quality compressed images. Simple Player™ is an application program that uses QuickTime™ to play and edit video sequences. Simple Player™ provides a diverse range of interactions, rather than a repetition of similar and simple behaviours (contrast this with a form-filling or a command-line interface). Simple Player™ was selected as the subject of the reverse engineering case study in fulfilment of the requirements listed below. These requirements had been set a priori for choosing the application-subject of the case study.

- The application should be easily available for experimentation. Intricate contingencies between interactive dialogues are difficult to foresee and have to be re-examined intermittently throughout the specification activity.

- The temporal behaviour of the application should be interesting and challenging to model. As a counter example, a form filling interface where fields are completed in any order does not have an interesting temporal element. In contrast, modification of the frame-rate of a movie while it is playing and the modes in which this activity is enabled is a more appropriate test for the model.

- Layout and graphical properties are not an important consideration in choosing the application. It is known that their description is not a strong point of the ADC model, since it does not incorporate an explicit model of the display.

- The size of the case study will be an indication that the ADC model is capable of scaling up: it is not sufficient to study a simple interaction technique or just one aspect of the behaviour of the system.

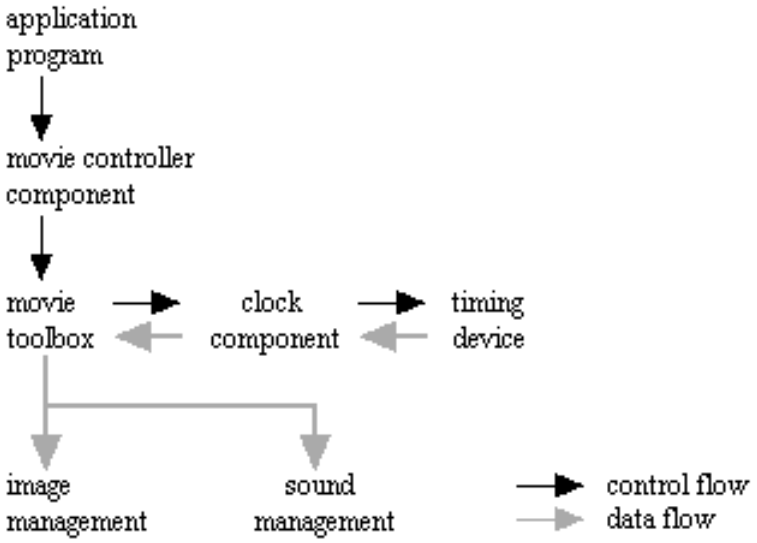


Figure 5.1. Relationships of an application, the movie controller component, the Movie Toolbox, and a clock component (adapted from [9]).

5.3 Some basic concepts of QuickTime™

QuickTime™ uses the metaphor of a *movie* to describe time-based data. The movie is a multiple-layer hierarchical organisation of data but the application that uses it need not be aware of this organisation. Simple Player™ may access the movie data through a set of *movie playback functions*.

An important element of movie data is the specification of the time dimension: at what rate will the movie be played, for how long, etc. A movie's *time coordinate system* defines a notional axis for measuring time, marked with a scale which defines the basic unit of measurement, the *time scale.* The time coordinate system specifies a *duration*, which is the length of a movie measured in time units. A point in a movie is identified by the number of time units elapsed from the beginning of the movie. The current position in a movie is defined by the movie's *current time*. When the movie is playing, this time value changes. A movie is characterised by the *rate* at which time passes for the movie. This specifies the speed and the direction in which time passes in a movie. Negative rate values move backward through a movie's data and positive values move forward.

Special clock components generate time information for the use of the Movie Toolbox. Clock components derive their timing information from some external source, e.g. some special hardware installed in the Macintosh computer to provide its basic timing. Figure 5.1 shows the relationships between an application, the movie controller component, the Movie Toolbox, and a clock component.

Other state information related to the movie is: the movie preview, the current selection, the active movie segment, the movie's display characteristics, preferred play-back volume, current volume, and preferred rate. The active movie segment is the part of the movie that the application is interested in playing. By default, it is set to the entire movie. It may be changed to some segment of the movie, for example, in order to play a user's selection repeatedly.

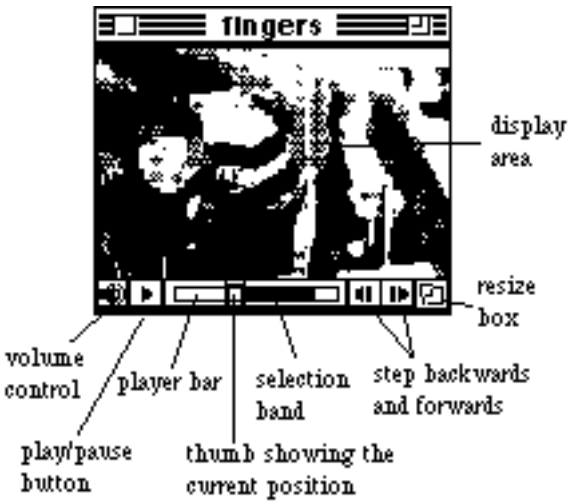


Figure 5.2. An instance of the Simple Player™ in operation. Some movie controller components are indicated.

Display characteristics are defined by a group of display regions, and a set of functions to operate on them. The Movie Toolbox hides the intricacies of handling the display characteristics of a movie, via the functions *GetMovieBox* and *SetMovieBox* which are used to display a movie at a particular location on the screen.

5.4 Interaction with Simple Player™

Simple Player™ supports a mixture of interaction techniques:

- With the standard Macintosh menu bar.
- Issue of commands by keyboard shortcuts.
- Graphical interaction techniques.

The ADC model is primarily focused on modelling graphical interaction, so the menu based interaction and keyboard commands do not concern this case study. The graphical interactions, which are the subject of the specification are described below:

- Setting the volume. A slider allows the user to adjust the sound volume. It is displayed when pressing the mouse button with the cursor over the volume control button. The user may change the sound volume while the movie is playing.
- Starting the movie. A play/pause button allows the user to start and stop the movie. Clicking on the play button causes the movie to start playing and the play button changes into a pause button. Clicking the pause button causes the movie to stop playing and the button to change back to its 'play' state. If the user starts the movie

and does not stop it, the movie controller plays the movie until its end. Playing may be achieved also by double-clicking on the movie image or, indirectly, by controlling the movie speed.

- Stopping the movie. The movie will pause by clicking on the movie image, or the play/pause button. It can be stopped indirectly by setting the movie speed to zero. The button is characterised by some *active area* surrounding it. If instead of clicking instantaneously the user drags the mouse out of the active area, the opposite effect than the mouse press is invoked. Dragging it in will repeat the effect of the mouse press.

- Controlling the rate of play. Pressing the mouse when the cursor is over the buttons with the right and left pointing arrows, while the 'control' key of the keyboard is pressed, causes a slider to 'pop-up'. This slider controls the play-back rate. Pushing the play button while the 'option' key is pressed will make the application play every single frame.

- Displaying a particular frame. Any frame of the video may be accessed randomly with the play-bar, or by stepping ahead or backwards using the buttons with the right and left pointing arrows. The play-bar is a slider that allows the user to navigate through a movie's contents. Dragging the thumb within the play-bar displays a single frame of the movie that corresponds to the position of the indicator. Clicking within the slider causes the indicator to jump to the location of the mouse click and causes the movie controller component to display the corresponding movie frame. The user may 'jump' to the beginning (or the end) of the movie with option-step forward (respectively backward). This will also interrupt playing.

- Defining a selection. If the 'shift' key of the keyboard is pressed, the current position in the movie will be used to define a selection. The selection is shown as a black band on the play-bar. The end-points of this selection may be changed by dragging the thumb, playing the movie, or stepping forwards and backwards through the movie while the 'shift' key remains pressed. The user may cancel a selection by clicking in the play bar away from the thumb.

- The user may drag the size box to change the window size, even while the movie is playing. Only the outline of the window is shown while this happens.

Simple Player™ provides just a small subset of the functions of QuickTime™. The interesting characteristic of the interface is that this group of functions is invoked in many ways. These are modal, so the effect of user input varies depending on the state of other interactions. Thus the temporal ordering of interactions is interesting to model.

5.5 Scope of the specification

The scope of the application of the ADC model is the Simple Player™ interface software only. This is depicted schematically in figure 5.3, mediating between a user and the

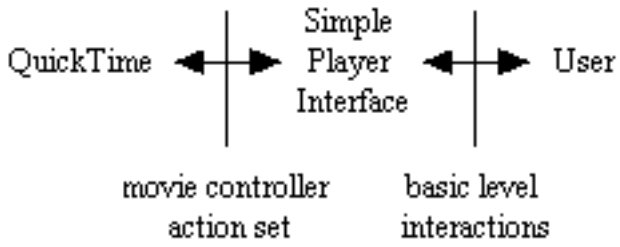


Figure 5.3. Scope and boundaries of the specification of the interface software.

QuickTime™ system extension. The left boundary, with the QuickTime™ functional core, is defined by the set of the *movie controller actions* which are a shorthand for accessing the Movie Toolbox function interface. In a few cases the interface software will access directly Movie-Toolbox functions.

The level of abstraction of input and output actions determines the right boundary of the scope of the interface. Input actions are considered as possible stimuli to the interface components and no assumption is made about their source. For example mouse actions over different interaction objects are considered independently as stimuli to the presentation objects they refer to. The event management mechanism that would relate these actions to their source, the mouse in this case, is not modelled. The same holds for output. There is no persistent and global model of the display contents during interaction.

5.6 A summary of the specification process and its product

The interaction with Simple Player™ was thoroughly studied through experimentation and using the on-line manual. An informal description of this interaction was assembled and consulted throughout the case study. Tests and comparisons with the specification led to improvements when inconsistencies and errors were revealed.

The second step was to decide upon the scope of the exercise which was determined as described in the previous section. The boundary of Simple Player™ with the functional core of Quicktime consists in a set of actions and movie controller functions which interface to the required functionality. The description of the complete set of movie controller actions and movie controller functions supported by Quicktime is described in [9]. The complete set of actions and functions that constitute through which Simple Player™ accesses the functional core is summarised in [124]. This set can be thought of as a ‘protocol’ describes the range of movie actions, their parameters and the output they produce. The temporal ordering of the invocation of these actions and functions was observed through Simple Player™.

The actions and functions accessed through the graphical interface constitute what was called the ‘functional core’. A formal specification of this core was produced. Each of the actions or functions of the application interface was mapped to a gate of the

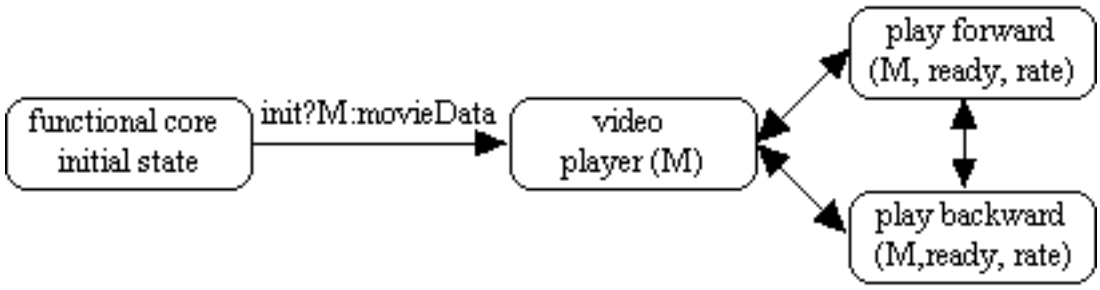


Figure 5.4. The operation of the functional core: Each node is described by a separate process specification. Arrows indicate invocations of a process by another.

functional core. The specification of the functional core used a data type specification of the movie data consistent with the description of [9].

Two versions of the functional core specification were studied: one modelling the passage of time units and the other modelling only the resulting non-determinism. The initial specifications of the functional core were improved and corrected by testing in conjunction with the interface specification. Both the timed and the untimed versions are interchangeable. The counting of time is transparent to the interface which is affected only by the resulting non-determinism. The two equivalent specifications of the functional core are outlined in [124].

Figure 5.4 illustrates the behaviour of the functional core as a state transition diagram. From the initial state, an initialisation action sets the local state variable M. This variable holds all the state information comprising the movie data. The movie may start to play forward or backward in which case the state of the application is also described by the current rate and a status flag ‘ready’. This flag determines if the application is ready to output a new frame or not. The states *videoPlayer*, *playForward*, and *playBackward*, shown as nodes in the diagram of figure 5.4, are specified as LOTOS process instantiations. This is an example of the state-oriented specification style discussed in section 3.8. In each of the states the functional core offers all the possible actions comprising the protocol of movie controller actions and functions. These will effect changes in the state variables and possibly transitions between the states of figure 5.4, when the play is stopped or its direction reversed. The termination of the program is possible from all states (termination is not indicated in figure 5.4).

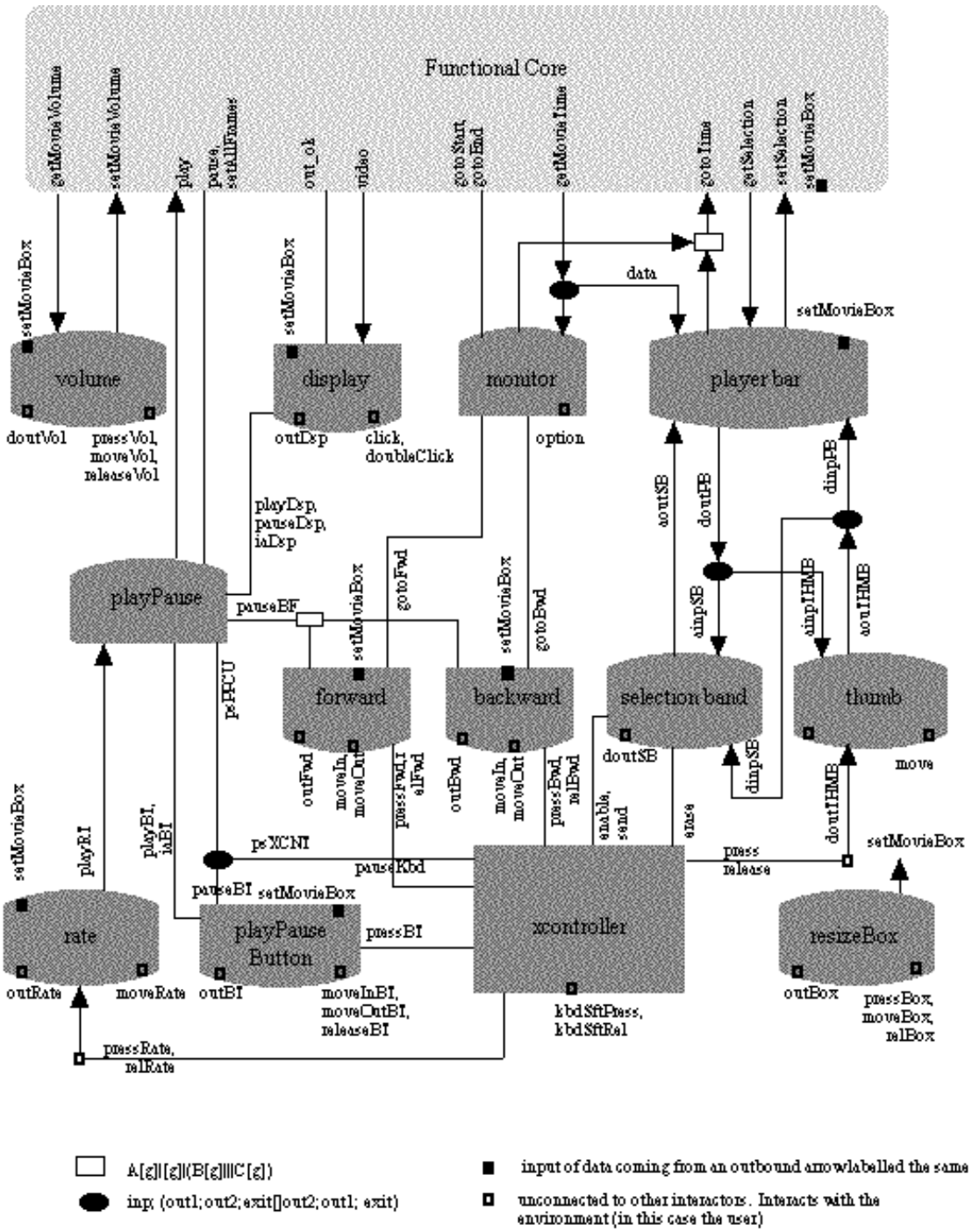


Figure 5.5. Graphical representation of the configuration of ADC interactors which model the Simple Player™ interface.

The specification of the user interface was structured as a graph of communicating interactors, marked with the data flows amongst them. This design consisted a working hypothesis as to how the interaction software could be structured in terms of ADC interactors. The defined structure is not realistic, i.e. it refers only to the specification. It is not at any moment supposed that the actual software is structured in this way. The architecture of the specification is not even optimal. It is possible, that alternative

configurations of interactors could do equally well or even better to model the interface, though this would be a subjective assessment.

Specifications of interaction components were individually tested and integrated into the specification one by one. During this process the initial structure of the specification was revised, with the addition of interactors that had not been foreseen initially, taking into account unexpected dependencies between interactors. The final configuration of interactors that comprise the specification is shown in figure 5.5.

The combined behaviour of the functional core and the interactors was specified by the synchronous composition of the corresponding processes. In this way the interactors constrain the temporal behaviour of the functional core, and vice versa. For example, the functional core may output a frame, only if the display interactor is ready to receive it. Gradually all gates of the functional core were so constrained. Interactors were added in a 'top to bottom' fashion, starting from those interacting directly with the application and finally specifying those controlled by the user. The interactors and their purpose are briefly summarised below.

The *volume* interactor models the pop-up slider used for volume control. Its display may have two forms: a slider when it is popped-up or a button when it is 'dormant'. Its abstraction is a bounded integer value representing the volume setting. It receives mouse input from its display side. It sends to the functional core a volume value at every movement of the mouse. The volume may be changed while the movie is playing.

The *resizeBox* interactor models a simple button that may be dragged. It outputs coordinate information on its gate *setMovieBox*. This value is read by the application, effecting the relevant changes to the display of the video and by all other interactors which are repositioned when the window is resized.

The *display* interactor models the window area that outputs the movie image. It receives video data from the functional core through gate *video*. It also synchronises with the functional core through gate *out_ok*. This allows the application to control the rate of play. The display interactor may be used to start and stop playing. The display interactor does not maintain any abstraction state, so it is a display-only interactor.

A group of three interactors the *PlayerBar*, the *selectionBand* and the *thumb* support all interactions related to the movie slider. The abstraction of *playerBar* contains a description of the movie duration, the current time in the movie and the current selection. Operations of the data type *playBar_ad* convert time information to geometrical information and vice versa. Interactor *selectionBand* constructs a line segment on the play bar display which corresponds to the selection. The *thumb* displays and lets the user manipulate the current time of the movie.

A second group of interactors is formed by the *monitor*, *forward* and *backward* interactors. The *forward* and *backward* interactors do not hold any abstraction value. They are push buttons that may give syntactical feedback when pressed. They may be used to 'step' forward or backward through a movie. With the right keyboard modifier

they effect a jump to the start or to the end of the movie but this is described in the monitor component. Interactor *monitor* does not have a display. It maintains an abstraction state parameter which is the current position in the movie. This is an integer which is incremented or decremented when it receives a command from the forward and backward buttons respectively.

Interactor *playPause* is another non-displayed interactor. It coordinates all interactors that issue a play or a pause command. Its abstraction state is an integer representing the rate of play. It has quite a complicated controller component since it manages many sources of play and pause actions.

Interactor *playPauseButton* is a display-only interactor. Its display is a simple two-state button but it has quite a complicated temporal behaviour.

Interactor *rate* is another pop-up slider. It sends its abstraction state which is a bounded integer value to the *playPause* interactor.

Interactor *xcontroller* defines the presentation-level dialogue for keyboard modifiers. Most interactors are affected by keyboard modifiers. The effect of keyboard modifiers is to enter and exit into modes. This is more naturally described in a state oriented style of specification. Interactor *xcontroller* constrains the behaviour of most other interactors and it is a controller-only interactor, i.e. it has no abstraction or display.

Finally a set of auxiliary interactors was introduced, shown as small ellipses or rectangles in figure 5.5. These are specified as abstraction-only interactors that support the data-flow between the components of the network. Two types are used in the case study. The rectangle means that any one of the sources of data may be used as input. The ellipse means that both receiving interactors will receive the data asynchronously. This class of interactors is discussed in section 5.8 as *logical connectives* that facilitate the composition of interactors.

5.6.1 Example: The specification of volume control

The volume control is displayed as a simple button when it is idle and as a slider when it is activated. A mouse press over the button will activate the interactor. The mouse press does not input any data to the interactor, but subsequent movement of the mouse will cause the position of the mouse to be interpreted producing a bounded integer value for the volume. This is the abstraction state.

The data type corresponding to the volume control has both abstraction and display sorts. Respectively these are a bounded integer and a graphical entity. The data input on the display side are screen coordinates (points). The data input and output on the abstraction side are integer values. The interactor may also receive coordinate information for its display from the abstraction side. The specification of *volume_ad* follows from the template for the AD data type definition introduced in chapter 4.

```
type volume_ad is popUpSlider
opns
    inputPr:        volumeBar, Int          ->      Int
    echoPr:         volumeBar, Int          ->      volumeBar
    inputMov:       pnt, volumeBar,Int      ->      Int
    echoMov:        pnt, volumeBar, Int     ->      volumeBar
    inputRel:volumeBar,Int          ->      Int
    echoRel:volumeBar, Int          ->      volumeBar
    renderRV:       volumeBar, rct          ->      volumeBar
    receiveRV:      Int, rct                ->      Int
    renderV:volumeBar, Int          ->      volumeBar
    receiveV:       Int, Int                ->      Int
    result:         Int                     ->      Int
eqns
forall r:rct, p:pnt, v, n:Int, vb:volumeBar
ofsort Int
    result(v) = v;
    inputPr(vb,v) = v;
    inputMov(p,vb,v) = pntToInt(vb,p);
    inputRel(vb,v) = v;
    receiveRV(v,r)=v;
    receiveV(v,n)=n;
ofSort volumeBar
    echoPr(vb, v) = popUpSlider(vb);
    echoMov(p, vb, v) = chIconAndSlider(vb,p);
    echoRel(vb, v) = popDownSlider(vb);
    renderRV(vb,r) = changeRect(vb,r);
    renderV(vb,v) = IntToBar(vb,v);
endtype
```

The data is input on the display side at gates *pressVol*, *moveVol* and *releaseVol.* The display of the interactor is output on the *outVol*. The abstraction side communicates directly with the functional core on gates *getVolume* and *setVolume*. Finally, it communicates with the *resizeBox* interactor on gate *setMovieBox*. The ADU specification for the volume interactor follows mechanically from the general definition of chapter 4.

```
process adu[press, move, release, dout, ainpR, ainpV, aout](a:Int,dc,ds:volumeBar) : noexit :=
    dout!dc; adu[...] (a, dc, dc) []
    ainpR?x:rct;        adu[...] (receiveRV(a,x), renderRV(dc,x), ds) []
    ainpV?x:Int;        adu[...] (receiveV(a,x), renderV(dc,x), ds) []
    press;              adu[...] (inputPr(ds,a), echoPr(ds,a),ds) []
    move?x:pnt;         adu[...] (inputMov(x,dc,a), echoMov(x,dc,a), ds) []
    release; adu[...] (inputRel(ds,a),echoRel(ds,a), ds) []
    aout!result(a);     adu[...] (a,echoRel(ds,a), ds)
endproc
```

The constraints component for the interactor *volume* should include constraints to describe:

- Dragging the mouse. A mouse press is followed by a sequence of moves, which is interrupted when the mouse button is finally released.

- Communication with the functional core. After a press the volume interactor reads the volume setting of the movie before popping up the volume slider. When dragged it gives feedback of the mouse position by moving the slider before setting the volume of the functional core.

```
process cc[press, move, release, dout, ainpR, ainpV, aout] :noexit:=
    triggers[press, move, release, dout, ainpR, ainpV, aout]
            |[press, move, release]|
    inp_control[press, move, release]
endproc

process triggers[press, move, release, dout, ainpR, ainpV, aout]: noexit :=
    press; ainpV?x:Int; dout?x:volumeBar;          triggers[...] []
    move?x:pnt; dout?x:volumeBar; aout?x:Int;      triggers[...] []
    release; dout?x:volumeBar;                     triggers[...] []
    ainpR?x:rct;                                   triggers[...] []
    dout?x:volumeBar;                              triggers[...]
endproc

process inp_control[press, move, release]: noexit :=
    press; (repeat[move][>release; inp_control[press, move, release])
endproc

process repeat[m]:noexit:= m; repeat[m] endproc
```

The controller unit for the volume control interactor follows from the general form by substituting the CC specified above.

5.7 The approach to specifying each interactor

Each interactor was specified after its connections had been fully worked out. The data handled by the interaction was specified trying to answer some simple questions:

- Does the interactor have a display?
- Does it provide input to the application or to other interactors?
- What is the sort of the data transferred through each gate?
- Is a gate destined for input, output or for dialogue actions only?
- What temporal ordering can the interactor induce on the actions on its gates?

These questions pertain to the external behaviour of the interactor only. However, they determine the specification of an interactor. If the interactor has a display, it will be equipped with the appropriate sort of data, and probably an input and an echo operation. If it sends data to other interactors it will be equipped with an abstraction state variable, receive and render operations to handle input and output data respectively. Once all gates have been assigned to either of the abstraction, display, or controller sides and once their typing has been determined, the definition of the ADU follows mechanically.

Data types were specified along with their corresponding interactors. The sorts of the display and abstraction data, and the typing of input and output gates determine the signature of the data type. If the interactor has an abstraction component it should at least have a receive and a result operation. If it also has a display status then an input operation uses the display status to interpret the input. It will also have an echo operation to update the display on the receipt of input, and a render operation to translate input from the abstraction side. In short, the signature of the interactor-specific data type *ad* is derived, almost mechanically from the sorts of the interactor's state parameters and the sorts of the data input and output. The semantic description of the functionality of a given interactor is specified by adding equations to this data type.

Interactors synchronise on their standard control behaviours. They share the corresponding part of their CU definition and synchronise at the SSRRA gates. Custom behaviour specifications are written as a synchronous composition of behaviours in the CC component using the constraint oriented specification style (see section 3.8). There is only one exception to this, which is the state oriented description of *xcontroller*. In the CC process definition, all LOTOS temporal operators may be used to combine behaviours, but as a matter of style enabling and disabling was avoided. The reason for this convention is that mixing parallel operators with enable and disable leads to infinite behaviours. Avoiding this kind of behaviour expressions facilitates the use of model checking tools.

Repeating patterns of temporal ordering constraints were identified across interactor specifications. This suggests that the specification exercise would be facilitated by a systematic re-use of such specifications. This issue is discussed more extensively in the following section.

5.8 Improvements to the ADC Model

The specification case study prompted modifications to the definition of the interactor model. These are:

- The introduction of dialogue actions for the controller and the modification of the ADU definition to support input actions that do not carry any data.
- The introduction of special cases of the ADC interactor that do not have an abstraction or a display, or that have no state component at all.
- The definition of a set of special purpose 'logical connectives'.
- The redefinition of the standard control behaviours.
- The identification of repeating patterns of temporal constraints.

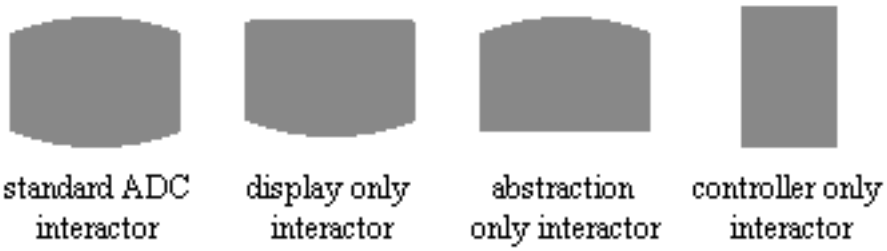


Figure 5.6. Diagrammatic representations for all types of interactors.

5.8.1 Non data-carrying actions

The original definition of the ADC model [123] stipulated that all actions over the gates of the ADU carry data and that the alphabet of the controller consists in the alphabet of the ADU plus the control actions *start*, *suspend* and *abort*. Two more classes of actions not accounted for in the original definition of the model are introduced:

- Actions that carry no data but are in the alphabet of the ADU. For example a simple synchronisation action on the gate 'erase' of the interactor *selectionBand* will erase the selection.
- Controller actions other than on the SSRRA gates. They are referred to as 'dialogue' actions as they are used to model temporal ordering constraints on the interaction.

The definition of the ADC model in chapter 4 has already incorporated 'dialogue' gates for the controller and the updated version of the SSRRA behaviours. Accommodating for actions that are not associated with the communication of data is only a syntactic improvement. However, it makes the resulting specification shorter and simpler.

5.8.2 Abstraction-Only, Display-Only and Controller Interactors

These are degenerate versions of ADC interactors. The *abstraction-only* interactor does not display any part of its state, e.g. the monitor interactor of figure 5.5. Consequently it supports only the *receive* and *result* operations of the general data type *ad* (section 4.3). A *display-only* interactor does not hold any abstraction value, e.g. simple push buttons like the forward and backward step buttons of figure 5.5. A display-only interactor only supports the operation *render* of the general data type *ad*. A *controller* interactor has neither of the two types of state component and models exclusively temporal ordering constraints, e.g. the *xcontroller*. It is a Controller Unit or a Constraints Component, as they were introduced in chapter 4. Instead of applying some temporal ordering on a single ADU, a controller component can be composed synchronously with any number of ADC interactors. In this way temporal constraints can be applied incrementally on a composition of interactors, using the constraint oriented specification style.

The concept of the interactor, as outlined in chapter 4, was associated with objects that have a display. It is though convenient to use interactors that do not have a display component as building blocks for more complex specifications. They can be used in combination with interactors which have a display component, when the interface is described as a composition expression. These 'reduced' interactors are illustrated as barrels, which are sliced along their axis. In each case the visual representation loses the arch that corresponds to the missing state component, as in figure 5.6.

5.8.3 Logical connectives

The term *logical connectives* is used here to describe a set of behaviour expressions that support the communication of data between interactors. LOTOS is a synchronous language so sometimes the communication of data between interactors may introduce unwanted dependencies. For example, considerations of modularity suggest that the sending interactor should not be 'aware' of a list of recipient interactors. Logical connectives are introduced with the aim to factor out such communication dependencies. Some are processes that buffer information between producers and consumers. These may be thought of as abstraction-only interactors. In other cases, the required communication scheme is supported directly as a LOTOS process algebra operator. Since one of the aims of the case study has been to investigate how interactors can be composed to model interface software, it is useful to reflect on what types of communications schemes may be needed.

Connections between ADC interactors represent data flow or pure synchronisation. In the case where data is communicated the ADC model distinguishes the producer from the consumer of the data. Communication schemes may be classified by the number of the producers and the consumers of data, and by whether they should all receive the data (AND) or just one of them should (XOR). AND related groups of producers can be specified in LOTOS as a set of processes synchronising on one or more of their output gates. The synchronisation of interactors on their output gates introduces the possibility of a deadlock (see table 3.1), when they offer inconsistent values. This possibility is rather contrived in the context of user interface specification, and blurs the distinction between the data handling component (the ADU) and the temporal ordering component (the CU) of the interactor. For these reasons, this type of connection between output gates is ruled out. The remaining possibilities are illustrated in table 5.1 along with their visual representation.

In table 5.1, the construction [g] denotes a set G such that g G. So when processes are combined in parallel over a gate list [g] it is implied that their synchronisation gate-set should at least include g.

Most of the communication schemes of table 5.1 are synchronous and they are specified directly as LOTOS behaviour expressions. The component *eventQ* is introduced to model asynchronous communication between interactors. It aims to free the producer and the consumers of data from constraints they would otherwise impose on each other

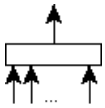
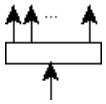
	one producer	many producers (XOR)
one consumer	P[g] |[g]| C[g] 	(P_1[g]|||...|||P_n[g]) |[g]| C[g] 
synchronous reception by many consumers (AND)	P[g] |[g]| (C_1[g] |[g]| ... |[g]| C_n[g]) 	(P_1[g] ||| ... ||| P_n[g]) |[g]| (C_1[g] |[g]|...|[g]| C_n[g]) 
asynchronous reception by many consumers (AND)	(P[g] |[g]| eventQ[g,g_1,..,g_n]) |[g_1,..,g_n]| (C_1[g_1] ||| ... ||| C_n[g_n]) 	(P_1[g] |||...||| P_n[g]) |[g]| eventQ[g,g_1,..,g_n] |[g_1,..,g_n]| (C_1[g_1] ||| ...||| C_n[g_n]) 
reception by one of many consumers (XOR)	P[g] |[g]| (C_1[g] ||| ...||| C_n[g]) 	(P_1[g] |||...||| P_n[g]) |[g]| (C_1[g] ||| C_2[g] |||...||| C_n[g]) 

Table 5.1. Logical connectives, their structure and their visual representation.

because of the synchronicity of LOTOS. The behaviour expression below suggests that some data x received on a gate g is communicated, in any order, on all of the gates g_1,..,g_n. Note, that this abstract definition for the event distribution scheme introduces a very high degree of parallelism into the specification.

```
process eventQ[g, g1, g2,...,gn]: noexit:=
g?x:data;  (send[g1](x)|||send[g2](x)|||...||| send[gn](x)) >> eventQ[g,g1,g2,...gn]
endproc

process send[g](x:data):exit:=  g!x; exit endproc
```

Many mechanisms for asynchronous communication can be imagined as plausible implementations of process *eventQ*, which is defined as abstractly as possible. These may be interesting in the context of implementing some user interface software. They are not as interesting for the purposes of its formal specification, so this issue is not discussed further. The set of expressions of table 5.1 are the minimum necessary general-purpose 'logical connectives' needed to compose interactors. Without their introduction an ADC interactor specification would have to be modified depending on the context of its use.

5.8.4 Temporal ordering constraints

During the case study, many similarities were observed across the constraints components of interactors. The identification and classification of common and re-usable expressions of temporal ordering constraints is essential for the practical application of the ADC interactor model. To scale up the use of the ADC model, it may be useful to construct a library of the corresponding process specifications. As a first step, some constraint expressions that recur across the interactors of the case study are discussed below.

Input Constraints

The term *input constraints* is used here to refer to LOTOS behaviour expressions used in the CC component, that involve gates that belong to the gate set G_{dinp} only. In the case study this concerns mostly mouse input. For example, after pressing the mouse button it will have to be released before it is pressed again. If mouse movement is allowed, e.g. when dragging is modelled, any number of moves may take place before the mouse button is released.

This is captured by the process description

```
process drag[press, move, release]:noexit:=
    press; (repeat[move]
[>  release; drag[press, move, release])
endproc

process repeat[m]:noexit:=
    m; repeat[m]
endproc
```

Push buttons use a simple variation of this constraint, where the movement is defined simply as moving in or out of the active area of the push button.

```
process push[press, moveIn, moveOut, release]:noexit:=
    press; (alternate[moveIn, moveOut]
[>  release; push[press, moveIn, moveOut, release])
endproc

process alternate[A,B]:noexit:=
    B; alternate[B,A]
endproc
```

When no movement is modelled, the constraint may be described as an alternation of press and release interactions:

```
process inp[press, release]: noexit :=
    alternate[press, release]
endproc
```

In the case study, only a few input behaviours were identified, since most interactions were achieved by clicking, dragging, etc. Clearly this set of constraints refers to the

syntax of the input language. Here, it refers to elementary actions at the interaction toolkit level. Similar syntactic regularities may be described as temporal constraint for more abstract descriptions of an input language.

Weak Feedback Constraints

Weak feedback constraints are temporal constraints, which relate input gates on both sides of the interactor with output gates on the display side. They describe how actions on the input gates of the interactor cause the occurrence of output actions on the display side. These constraints are called weak because they do not force the output immediately after the input. Rather, they prohibit new input from being accepted before an output event. In the example below, an action on any of the input gates of the interactor is prohibited until the state of the interactor is shown to the user.

```
process weak[dinp, ainp, dout]: noexit:=
    dinp; dout; weak[dinp, ainp, dout]
[]  ainp; dout; weak[dinp, ainp, dout]
endproc
```

The gate set of the process contains only input and output gates directly concerned with the feedback constraint. The remaining gates of the interactor are therefore unconstrained. This is achieved by synchronising with the process SU_RE of the controller unit only on the common gates of the constraint. When using the simple model without the parameterised behaviours, the constraints apply directly to the process *adu* as follows:

```
adu[dinp, dout, ainp, aout]
    |[dinp, ainp, dout]|
weak[dinp, ainp, dout]
```

Consider for example a push button where the input gates on the display side are *press*, *move* and *release*. This constraint would be written as:

```
process buttonFeedback[press, moveIn, moveOut, release, dout]:noexit:=
    (choice g in [press, moveIn, moveOut, release]
    [] g; dout; buttonFeedback[press, moveIn, moveOut, release, dout] )
endproc
```

These temporal ordering properties of the behaviour of an interactor or an interface should not be confused with closely related notions such as fine-grain semantic feedback or observability, etc. The notion of adequate feedback depends also on the information content of the output. Whether the information displayed is actually visible or not depends also on the ergonomic design of the display, the user capabilities, etc.

Triggers and Strong Sequencing Constraints

Consider the specification of a trigger behaviour in the constraint oriented style, as was demonstrated in section 4.5. To specify the trigger behaviour, it is important to constrain

all the gates of the CC component, to ensure that after a trigger action the required behaviour will follow. Triggers are a special case of the more general constraint where only a choice between certain sequences may be observed. Once a sequence is entered all other interactions are prohibited. This is why these constraints are called strong sequencing constraints. For example, after a user input it may be required that an echo follows immediately, and after the echo an output on the abstraction side should follow. In this way, strong feedback constraints may be specified, relating input gates and output gates of the interactor.

As an example consider the triggers constraint for the step-forward button. It ensures that the command *goto* is issued before any other interaction.

```
process triggers[press, moveIn, moveOut, release, goto, ainp, dout] : noexit:=
    press;                goto;                triggers[...]
[]  moveIn; goto;                 triggers[...]
[]  moveOut;                                   triggers[...]
[]  release;                                   triggers[...]
[]  ainp?x:rectangle;                          triggers[...]
[]  dout?x:twoStateButtonDsp;                  triggers[...]
endproc
```

To ensure that the trigger behaviour occurs all gates of the CC are included in the gate set of this process. So while only the first two lines of the behaviour expression specify the required triggering behaviour, clauses have to be added to ensure that the remaining gates may continue to interact. The alternative sequences of interactions are specified in the action prefix form. This is an example of the monolithic specification style discussed in section 3.8. Alternative sequences of actions, such as a trigger construct are easier to specify in this style.

Toggles and Modes

A double click on the image area of Simple Player™ effects a play command and a single click effects a pause command, when the movie is playing. The interactor toggles between two states depending on the state of the application, whether it is playing or stopped. For example, in the specification of the case study, the interactor *playPauseACU* informs the interactor *display* and the other sources of play and pause commands, as to whether the application is playing or not.

The toggle process is just one example of a constraint that maps naturally to the notion of state transitions. The toggle has two states, which are characterised by whether it offers action A or B. In each state, a toggle action will cause it to switch between states. After an action A or B takes place only a toggle action T can be accepted to complete the transition:

```
process toggle[T,A,B]:noexit:=
    A; T; toggle[T,B,A]
[]  T; toggle[T,B,A]
endproc
```

In the specification of the display interactor the toggle constraint was instantiated as

toggle[ainp, doubleClick, click]

while for the play pause button it was instantiated as

toggle[ainp, play, pause]

In summary, this section has discussed a small set of temporal constraints that are common across the interactors of the case study. These constraints are specified as LOTOS processes which are used in the constraints component of the ADC interactor. The specifications of this section have all been written in basic LOTOS. This is appropriate because these descriptions concern only the temporal ordering of action occurrences and not the information content of an interaction. However, the ADC interactor is specified in full LOTOS. The gates of the ADU are typed, so any specification of the temporal ordering of its actions has to take into account this typing. The systematic re-use of the constraint definition puts a requirement of *polymorhism* on their instantiation. Polymorhism refers to the ability of an entity to refer at run-time to instances of various types [131]. However, here the term is used to denote the ability to write constraints independently of the type of the data they apply to. Unfortunately, LOTOS does not support such a feature.

5.9 Assessment of the study: lessons drawn and limitations

The case study was first and foremost a useful experience in using the ADC model. Various graphical interaction styles were modelled. The ADC model turned out to be a useful framework for guiding and structuring the specification activity. The subject of this specification is a relatively small scale application. However, the specification is very detailed and has a high degree of parallelism, so the specified behaviour is quite complex. The specification was simulated on the Lite toolset [30, 119] and the simulation was compared with the behaviour of the Simple Player™ software. Model checking tools were used to verify some dialogue properties. The discussion on the analytical use of the model is deferred for chapter 7. In practice model checking had limited success because of the size and the complexity of the specification.

As an indication of the size of the specification, the text written was approximately 1500 lines of full LOTOS code. The complexity of the specification was compounded since the specification exhibits a much higher degree of parallelism than the software it models. The interface is described as a parallel composition of entities, which is not necessarily the case with the actual software. Communications between these components are not prescribed to take place in any specific order as this cannot be observed by experimentation with the actual software. This means that the reverse engineered specification has many more ‘degrees of freedom’ than the actual software. Though it is a correct specification of the observed behaviour, the high degree of parallelism puts the complexity of the specification beyond the capability of the tools

used. This problem persisted even after the specification was transformed to basic LOTOS for verification purposes.

The resulting specification is under-constrained for one more reason. The network of interactors in figure 5.5 does not describe end-to-end constraints, e.g. between gates on the application interface and the user side. In principle, such constraints can be built into the specifications of each interactor although this is detrimental for the modularity of the specification. On a methodological note a bottom-up and piecemeal study of the interface is not prone to reveal constraints relating the behaviours of interactors that do not communicate directly with each other. For example, consider the constraint that no user input should be possible before the current video image has been displayed. Such a constraint is easy to model, by the addition of a 'global' constraint component connected to the relevant gates.

Already, the experience of the case study suggests that it is necessary to update, or to extend, the original hypotheses regarding the use of the model. An interface specification is not built only by the composition of basic interactors. Stand-alone controller components and the logical connectives to support the data flow between them have to be added.

The use of the interactors is greatly facilitated by the existence of an overall architectural model. One of the main difficulties in this study was to define what is the application interfaced to. Defining a behavioural specification of the functional core is a useful activity that defines the boundary for the use of the interactor model. It requires adopting an overall architecture like for example the Arch reference model for interactive systems [14], discussed in chapter 2. Similarly the boundary of the scope of the interactive software needs to be defined at the display side, possibly by a display model.

Data type specifications for the interactor specifications follow directly from the general data type *ad*, by syntactic substitution of the actual sort identifiers. However, such specifications define only the syntax of the operations involved. To relate more to the domain in question it seems a promising idea to provide a library of abstract data types. This has been done in other fields where formal specifications have been applied, indeed LOTOS is packaged with a standard library for abstract data types, e.g. integer, Boolean, etc., (see [117]). The ADC model provides a framework for identifying re-usable data types and for using them in the specification of user interfaces. Depending on what kind of reasoning will be done with the resulting specifications, different models may be adopted for the display and the abstraction.

The case study has tested several aspects of the ADC model. However, some parts of it remain untested as a result of the choice of specificand. The Simple Player™ application, provides an interaction rich in temporal dependencies but the structure of the interface itself is quite static. It involves the same set of interactors throughout its operation. As a counter example consider a drawing package where dynamic creation/destruction of interactive objects is required. To model such objects by ADC interactors it is necessary to introduce a notion of dynamic creation and deletion of

components, which is more akin to objects in object oriented programming [131] than to the concept of processes. The interaction with Simple Player™ does not include navigation dialogue as is commonly found in hypertext systems, or form filling interfaces. Clearly, it is a worthwhile exercise to apply the model in this domain. This would legitimise claims that the model is helpful in the specification of such systems. However, there is no reason to believe that this class of system could not be described sufficiently using the ADC model.

Simple Player™ was modelled by a composition of interactors. Groups of such interactors were closely related and indeed they could be thought of as one. By the same argument a different specification approach would be to model the whole graphical interface, possibly at a more abstract level, as a single interactor. Indeed, this has been the intuition of using the model all along. In other words, some notion of congruence is required, by which the specifier will be allowed to substitute compositions of interactors with a single interactor. It should then not be necessary to refer to the original components when using the resulting composite interactor, analytically or as a component for further compositions.

During the specification exercise when the specification of a particular behaviour was hard to describe as a single interactor, the problem was divided into the definition of two or more communicating interactors. This was the case with the *playerBar* group of interactors and with the *monitor* group of interactors. Broadly, given that the example was reverse engineered, the specification was constructed bottom up. In a forward engineering example, it is very likely that the starting point would be an abstract specification of a higher level interactor, that would be refined by its decomposition into lower level entities.

This problem of switching between the two models of a user interface, as a single monolithic ADC interactor or as a structured collection of ADC interactors, are discussed in the next chapter as formal transformations. Other transformations are discussed too, that concern the effect of hiding the details of communication across interactors and the way in which standard dialogue behaviours are supported.

5.10 Conclusions

The case study showed the ADC interactor model to be an appropriate formalism for the specification of the graphical interaction with Simple Player™. The model provides a conceptual framework for the specification, i.e. it prompts a specification technique by which many difficult decisions are arrived at almost mechanically. However the original concept by which the interface is composed of ADC interactors only, was found to be restrictive. Modifications were introduced to the model to improve the modularity of the resulting specifications. The extensions include the simplified specification of ‘degenerate’ cases of ADC interactors and the classification and standardisation of special purpose components managing the data communication between interactors.

However, these extensions are simply ‘syntactic sugar’, since all these components may be specified as instances of the ADC interactor of chapter 4.

The case study was a positive but not definitive experiment. It does not provide a scientific assessment of how useful the ADC interactor model would be to practitioners, but rather it is a formative assessment of the formal model which will help improve it. Some aspects of the model have not been tested, and further experimentation with different types of interface software would be a useful continuation of this work. Nevertheless, the case study provides a concrete and far from trivial application of the ADC formal interactor model, which is a necessary step towards its further development and dissemination.

Chapter 6

Synthesis, Decomposition and Abstract Views

This chapter investigates some important and practically useful properties which have been built into the ADC model. To this point they have only been described intuitively and constitute hypotheses about the model. An important property of the model is its compositionality. In the present context, this term means that any LOTOS behaviour expression involving ADC interactors may be shaped into the form of an ADC interactor, while at the same time preserving its meaning. The implication is that an interactive system may be iteratively formulated as a single ADC interactor. To formalise this idea a more rigorous definition of the ADC interactor model is called for. Two syntactical transformations are defined, the synthesis and the decomposition of ADC interactors. These transformations preserve the meaning of the specification up to strong bisimulation equivalence [133]. The synthesis transformation was introduced in [125] and the decomposition transformation was outlined in [126]. These transformations are presented with their theoretical foundation and a reflection on their practical implications.

The concept of *abstract views* of interactors is also introduced. It refers to an interactor some of whose gates are hidden. Abstract views are useful for abstracting away from internal detail of composite interactors. The synthesis and decomposition transformations apply to abstract views as well. The effect of synthesis and the decomposition transformations on the standardised behaviours, which have been collectively referred to as SSRRA in earlier chapters, is examined. Finally, this chapter deals with the question of dialogue representation in the ADC model, extending some of the ideas outlined briefly in [126].

6.1 Rigorous definition of the ADC interactor

The ADC interactor model has been defined, in chapter 4, as a template for a LOTOS process definition associated with an abstract data type. A more abstract and rigorous description is required which will define unequivocally which LOTOS behaviour

expressions may be called ADC specifications. The approach adopted below examines the syntactic structure and the semantic requirements which characterise ADC interactor specifications in the LOTOS formal specification language.

6.1.1 Topology of interactor gates

As a first step towards a more abstract and general definition of the ADC interactor model, the formal model is extended to relate sets of gate identifiers (gate sets for short) rather than individual gate identifiers. This idea has been introduced already in chapter 4, where the set G_d = {it, ot} was used in the place of a single dialogue gate d, to model the trigger behaviours of the Pisa interactor model.

Gate lists rather than gate sets are the standard construct of LOTOS. However the discussion that follows describes process instantiations using sets of gates. A gate is described by its name. The manipulations and definitions of behaviour expressions that follow attach no significance to the order in which gates are declared in a process instantiation. In practical terms, to support this convention using LOTOS, the recursive process instantiations of the ADU and the CU should preserve the naming and ordering of gates in the process headings.

For example consider the process definition whose heading is:

```
process ADU[dinp, dout, ainp, aout] : noexit :=
```

The convention introduced for the definition of process ADC means that it will not be allowed to write a recursive instantiation of the form

```
ADU[dout, dinp, ainp, aout]
```

With the exclusion of such permutations the gate set G_{io}={dinp, dout, ainp, aout} describes sufficiently the gate-list for the process ADU.

The remainder of this paragraph describes the topological constraints on the gate sets with which an ADC interactor or its component processes are instantiated. The set G of the gates of the interactor is partitioned into the set of input-output gates G_{io} and the control gates G_c.

$$G = G_c \quad G_{io} \text{ and } G_c \cap G_{io} = \emptyset \qquad 6.1$$

Controller gates may effect standardised control behaviours which so far have been called the SSRRA behaviours (respectively the formal gate identifiers). Alternatively they may effect actions carrying no data which are handled by the controller component. The subsets of G_c are mutually exclusive.

$$G_c = \text{SSRRA} \quad G_d \text{ and SSRRA} \cap G_d = \emptyset \qquad 6.2$$

$$\text{SSRRA} = \{\text{start, suspend, resume, restart, abort}\} \qquad 6.3$$

G_{io} is partitioned to two gate sets G_{abs} and G_{dsp} that correspond to gates on the abstraction and display side of the interactor respectively.

$$G_{io} = G_{abs} \quad G_{dsp} \text{ and } G_{abs} \cap G_{dsp} = \emptyset \qquad 6.4$$

$$G_{abs} = G_{aout} \quad G_{ainp} \text{ and } G_{aout} \cap G_{ainp} = \emptyset \qquad 6.5$$

$$G_{dsp} = G_{dinp} \quad G_{dout} \text{ and } G_{dinp} \cap G_{dout} = \emptyset \qquad 6.6$$

The example below demonstrates the generalisation of the interactor model for the case where the gate sets are not atomic.

Example

Consider an interactor representing a file icon on a desktop interface. The file icon can be manipulated directly, i.e. it may be dragged and dropped. Suppose also that a double click of the mouse button when the mouse-cursor is over the icon fires an associated application, e.g. a word processor. One possibility is to model the file icon as an interactor with two input gates on its display side G_{dinp}={drag, drop} and an output on the display side G_{dout}={dout}. This interactor receives semantic information on its application side from a higher level entity that models the file system. The interactor will use such input to update its contents and its presentation to reflect the state of the file manager, e.g. date and time associated with the file, its name, etc. The file icon has an output on the abstraction side, through which it sends data to interactors modelling other entities, e.g. folder icons and the desktop interface. This data is described below as a value of sort *fileInformation*. The double click is modelled by a dialogue event G_d ={doubleClick} that synchronises with the start of another interactor modelling the window with the file contents.

Here, the ADU offers a choice of two input events on the display side:

```
process ADU[drag, drop, dout, ainp, aout] : noexit :=
    drag?x:position;              ADU[...](inputDrag(abs, ds, pos), echoDrag(abs, ds, pos))
[]  drop?x:position;              ADU[...](inputDrop(abs, ds, pos), echoDrop(abs, ds, pos)
[]  dout!dc;             ADU[...](abs, dc, dc)
[]  ainp?x:fileInformation;      ADU[...](receive(abs,x),render(abs,x), ds)
[]  aout!result(abs);  ADU[...](abs, dc, ds)
endproc
```

The other components of the interactor are almost the same as for atomic gate sets. They include one more gate in their gate set and the CC has to constrain this as well. For this example, let the only constraint be that feedback is given before any new input can be accepted, and that the file icon will send its result (suppose it is a file identity) only after receiving input on the abstraction side from another interactor.

```
process CC[drag, drop, doubleClick, dout, ainp, aout] : noexit  :=
    drag?x:position; dout?x:disp;                                  CC[...]
[]  drop?x:position; dout?x:disp;                                  CC[...]
```

```
[]  dout?x:disp;                                              CC[...]
[]  ainp?x:fileInformation; dout?x:disp; aout?x:fileInformation;  CC[...]
[]  doubleClick;                                              CC[...]
endproc
```

6.1.2 The set of possible interactions

The topology of the gate set has identified five sets which partition the gate set of an interactor. These are G_{dinp}, G_{dout}, G_{ainp}, G_{aout} and G_c. Each gate set corresponds to what can be called a different *role* of a gate for the interactor. This means, that the membership of a gate determines its use in the ADU and the CU. The definition of ADC in the following paragraphs, specifies actions on a particular gate according to which of the gate sets above it belongs to. For convenience, the role of a gate will be referred to simply as *dinp*, *dout*, *aout, ainp* and *c*.

Apart from these gates, the set of actions that may be offered by an interactor depends on the data offered on its G_{io} gates. This set of actions L is:

$$L \quad G_c \quad G_{dinp} \times dInpData \quad G_{ainp} \times aInpData \quad G_{dout} \times dsp \quad G_{aout} \times aOutData \qquad 6.7$$

where the sort identifiers of the data type AD, defined in chapter 4 indicate the domains of the data that may be communicated at an interactor gate.

6.1.3 The syntactic structure of ADC interactors

An ADC interactor is a non-terminating process (i.e. it has functionality *no-exit*). In general the process ADC is formed by the parallel composition of the ADU and the CU.

```
process ADC[Gc  Gio]: noexit :=
    ADU[Gio](a,dc,ds) |[Gio]| CU[Gc  Gio]
endproc
```

Following the case study of chapter 5, it was suggested that in some cases the ADC interactor should comprise of a CU only. In some cases it might even be desirable to reduce the CU to its constraints component (CC). Using a CC that does not support the standard behaviours results in simpler and more abstract specifications. In the following sections which describe the synthesis and decomposition of interactor specifications, the internal structure of the CU component is not of concern, so the results that follow hold both for the CU that supports the SSRRA behaviours and for a simple CC component as well.

6.1.4 The AD data type specification

Consider the example of the scrollable list examined in chapter 4. In the original synchronous composition expression, both interactors were associated with a custom data type: *scr_ad* and *ls_ad*. The composite interactor formed as a behaviour expression

that combines the two interactor instantiations, was rewritten into an equivalent form with the general syntactic structure of the ADC interactor. However, the composite interactor was not associated with a higher level data type AD combining the data types of the components.

In general, it is not necessary that an ADC interactor specification should be associated with an AD data type even if it models data communication. What is important for the discussion is that interactions on output gates actually offer a value and that interactions on input gates do not refuse a value offered to them. By the definition of the ADU of chapter 4 value offers may be either the *dc* state variable or the interpretation of the abstraction by the inquiry operator *result*. These operations are defined for the data type associated with the elementary interactors. It is required that elementary interactors are associated with an internally consistent data type AD, whose definition is formed according to the template of chapter 4. This ensures that the enquiry operators will always return a value.

As discussed already an interactor might have multiple input gates or output gates on either side. In most cases this requires the definition of a corresponding set of operations. The data type definition reflects the topology of the gates of the ADU:

1. For each gate in G_{dinp} an input and an echo operation have to be defined.

2. For each gate in G_{ainp} a receive and a render operation have to be defined.

3. For each gate in G_{aout} a result operation has to be defined.

4. For each gate in G_{dout} a display sort must be defined.

It is not necessary that different operations are defined for each gate. For example, there might be two input gates on the display side using the same input operation. However, a methodological guideline that can be postulated here is that if two gates are associated with the same operations, then they should probably not be distinguished. Their separation might be due to architectural considerations external to the interactor, e.g. the same data is communicated over the gate to two client interactors. This is detrimental for the modularity of the interactor specification and it is better that this distribution of data be supported by one of the logical connectives discussed in chapter 5.

In conclusion, only elementary interactors are associated with an AD data type. This data type is defined as in chapter 4 with the extra requirements listed above. Note, that these requirements do not exclude the possibility that an elementary interactor is related to more than one data type, or that it can be an abstraction-only or display-only interactor.

6.1.5 Elementary ADU

An ADU is a recursive non-terminating process for which the following holds:

1. Its gate set G_{io} can be partitioned in a set of input gates G_i and a set of output gates G_o, such that $G_{io} = G_i \quad G_o$ and $G_i \cap G_o = \emptyset$

2. Interactions on input gates that involve the communication of data are strictly variable declarations without a selection predicate (see section 3.8). Simple events that carry no data are also allowed as input.

3. Output actions are strictly value declarations. They are either the value of a local parameter, e.g. the display state, or the value of an enquiry operator on the data type AD, e.g. the result operation. In order that a value is always offered the data type AD should be internally consistent.

4. The elementary ADU is a behaviour expression that offers a choice of events on all the gates of the ADU before recursively instantiating itself. The recursive instantiation updates the local variables by applying the operations of the AD data type corresponding to the role of each gate.

Note that in accordance to the findings of the case study it is useful to model interactions on input gates that do not carry any data. These interactions have the effect of applying the operations specified in the ADU without communicating any data. Further, the set of interactions offered at each gate is not empty. Since AD is consistent there will always be some value output on the output gates, and by definition a range of interactions will be offered on input gates. The recursive instantiation of the ADU supports the mapping of gates to operations. So for example, for each gate in G_{dinp} there should be an input and an echo operation, etc. as discussed in section 6.1.4.

6.1.6 A well formed ADU

A well formed ADU may be:

1. An elementary ADU.

2. A parallel composition expression of the form

$$ADU_A[G_{io}^A]|[G]|ADU_B[G_{io}^B] \qquad 6.8$$

where ADU_A and ADU_B are themselves well formed ADUs, and

$$G_o^A \cap G_o^B \cap G = \qquad 6.9$$

3. The following is stipulated for the gate set of the well formed ADU:

$$G_{io} = G_i \quad G_o \text{ where } G_o = G_o^A \quad G_o^B \text{ and } G_i = (G_i^A \quad G_i^B) - G_o \qquad 6.10$$

The parallel composition of two ADUs that synchronise on some of their gates is also a well formed ADU. Condition 6.9 ensures that it is not possible to specify an ADU that might deadlock because of its two components offering different values on the same

output gate. Equation 6.10 describes the definition of the gate sets for the composite ADU. A gate which is an input gate for one of the components and an output gate for the other is classified as an output gate for the compound interactor. This ensures that the input and output gates of the derived ADU do not intersect, i.e. $G_i \cap G_o = \emptyset$, in accordance to the definition of the topology of the interactor gates. Further, it is clear that $G_{io} = G^A_{io} \quad G^B_{io}$.

Lemma 6.1. Behaviour of a well formed ADU

The behaviour of a well formed ADU is characterised by the following:

1. The set of interactions offered at each gate is not empty.

2. P(ADU) ~ Q, with Q = choice g in [G_{io}] [] g;Q

where ~ denotes strong bisimulation equivalence (see appendix A.1) and P(.) denotes the naive transformation from full LOTOS to basic LOTOS, which maps each specified interaction g<v> to some interaction g on the same gate but ignoring the data component of the action specification (see appendix A.1).

Process Q is defined so that it is always ready to offer interactions on all its gates. It is defined as a basic LOTOS process. Property 2 states that if the data values associated with interactions on the gates of the ADU are ignored, then its behaviour may be described by Q. This property is a rigorous expression of the fact that the ADU does not model the temporal behaviour of the interactor.

Proof.

By structural induction. By definition both properties hold for an elementary ADU. Property 1 holds trivially for the induction hypothesis since no selection predicates are introduced with the parallel composition and because of the requirement 6.9. Property 2 holds by the distribution of the naive transformation over the parallel composition operator [118, chapter 13].

Å

6.1.7 Specification of the CU

The temporal ordering of the interactions of the ADU is described in the constraints component CC. This is a recursive process with functionality *no-exit*, and it can be any LOTOS behaviour expression with gates $G_{cc} = C_{io} \quad G_d$. There are no further behavioural requirements for the CC.

The CU is defined as a template in which the CC is incorporated, so as to support the SSRRA behaviours. The presentation of this structure is deferred until section 6.5. For

the current discussion it is sufficient to consider an elementary CU as a process such that:

1. Gates in G_{io} are associated with variable declarations whose sorts are defined by the typing of the gates in the ADU.
2. CU does not have any local state variables.
3. CU does not output any values, i.e. it does not specify interactions with a value specification component.
4. CU does not have any selection predicates on actions specified with a variable declaration.

The CC satisfies these properties also, so for the purposes of the synthesis transformation it is possible that a CC is used in the place of a full CU definition. The definition above does not constrain the required behaviour of the CU. As will be shown in later sections the CU encodes the 'dialogue' supported by the interactor. Rules 1-4 restrict the use of LOTOS constructs, effecting the ease with which a dialogue may be specified. These restrictions do not diminish the expressive power of the notation which is that of Turing machines, as has been shown in [67].

6.1.8 Conclusion

In the start, this section set out to define a simple criterion by which a LOTOS behaviour expression could be determined to be an ADC interactor specification. It is now possible to introduce a formal definition of the ADC interactor:

Definition. ADC interactor specification.

A LOTOS behaviour expression is an ADC interactor specification if:

1. It is a well formed CU or
2. It is formed as the synchronous composition of a well formed ADU and a well formed CU, where the two processes synchronise on all the gates of the ADU.

6.2 Synthesis

As has been mentioned already it is possible to specify a complex user interface by the composition of ADC interactor specifications. The composition is a behaviour expression that uses the standard LOTOS process algebra operators. The property of compositionality means that this expression can be rewritten into a single compound ADC interactor, which satisfies the definition of paragraph 6.1.8. This rewriting transformation is termed *synthesis*.

The synthesis transformation is described by:

1. An input expression called *the distributed form* DF = ADC_A ADC_B where is a binary composition operator for non-terminating LOTOS behaviour expressions (one of |||, ||, [>, [] and |[G]| where G $G_A \cap G_B$).

2. An output expression called *the compound form:*

$$CF = ADU_{AB}|[G_{io}^{AB}]|CU_{AB}$$
$$\text{where } ADU_{AB} = ADU_A|[G_1]|ADU_B \text{ and } CU_{AB} = CU_A \quad CU_B \qquad 6.11$$

   The compound form is sufficiently defined by , G_{io}^{AB}, G_1.

3. The *transformation requirement* is that the compound form is itself an ADC interactor. By the definition of section 6.1.8 the expression CF is an ADC interactor, provided the ADU_{AB} is a well formed ADU. It is therefore necessary that:

$$G_1 \cap G_o^A \cap G_o^B = \qquad 6.12$$

4. The *correctness preservation requirement* describes the invariant of the transformation, i.e. it expresses the desired relationship between the semantics of the distributed form and the compound form. Clearly some congruence relation is required, i.e. the two forms should not be distinguished in the various algebraic contexts they might appear in [133, chapter 7]. However, as it is discussed more extensively in chapter 7, the comparison of behaviour specifications should reflect how they are perceived to be different or similar by a human observer and in the context of a user interface architecture.

Further debate as to how behaviours of interactors should be compared is deferred until chapter 7. In the next sections, strong bisimulation equivalence is proved between the DF and the CF for all possible behaviour composition operators . This is the strongest useful (i.e. excluding equality) requirement for comparing behaviours modelled as labelled transition systems [133]. The bisimulation equivalence of DF and CF entails the other weaker equivalence and congruence relations.

6.2.1 Synchronous composition of interactors

Static composition operators are examined first. Consider two interactors ADC_A and ADC_B with respective gate sets G_A and G_B. The operator |[G]| describes the partial synchronisation of ADC_A and ADC_B over a gate set G $G_A \cap G_B$. Full synchronisation || and pure interleaving |||, can be thought of as boundary cases of the partial synchronisation of behaviour expressions, where G=G_A G_B and G=Ø respectively.

In section 6.1.2 it was mentioned that each gate is assigned a role in an interactor. This may be the role *dinp, dout, ainp, aout,* or *c.* A gate g G may have different roles for

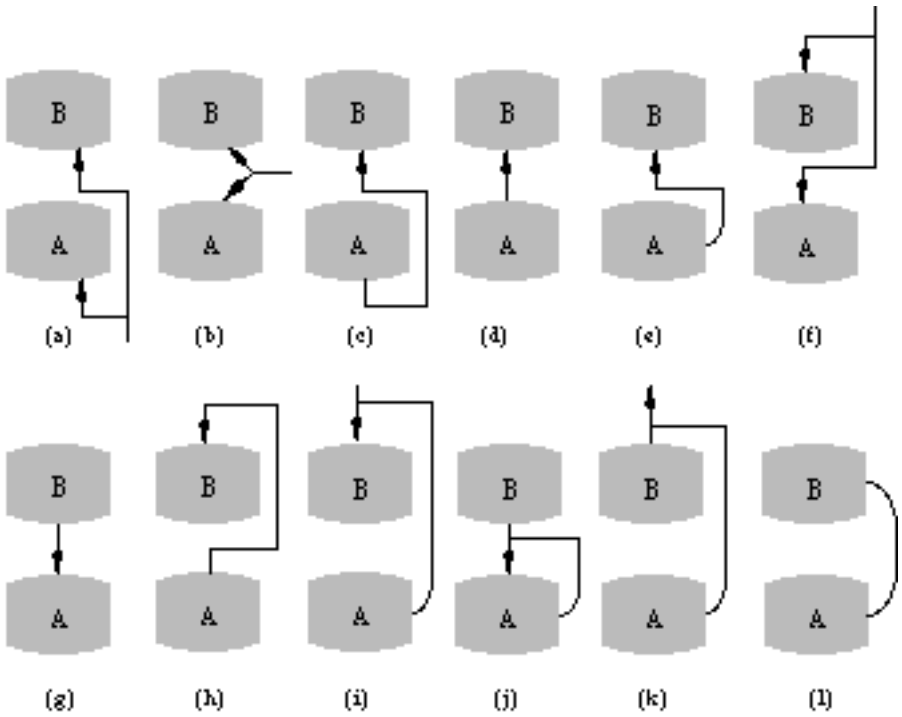


Figure 6.1. The range of connection types of ADC interactors. Arrows indicate gates for input or output. The last case represents a simple synchronisation.

each of the two interactors. A different *type of connection* is obtained for each combination of roles, e.g. (*dinp*, *dout*). This paragraph discusses these different types of connections and introduces some conventions for the synchronous composition of interactors. Since this discussion does not distinguish between interactors ADC_A and ADC_B, a connection type is symmetric i.e. the same type of connection corresponds to the combinations (*dinp*, *dout*) and (*dout*, *dinp*).

A composition of two interactors may use from none, in the case of pure interleaving, to many types of connections (a)-(l) below.

(a) Connection type (*dinp*, *dinp*). Both interactors receive data synchronously from their display side. In section 5.8 this was described as a case of multiple consumers receiving data synchronously. An example of this could be a multiple selection of icons which are 'dragged'. In this case the mouse position is read by both interactors.

(b) Connection type (*ainp*, *dinp*). The two interactors are synchronised consumers. In this case though, the input arrives at the display side for B and the abstraction side for A.

(c) Connection type (*dout*, *dinp*). This could be a case of the reuse of the graphical output of interactor A as graphical input to interactor B. For example, an interactor

may ‘capture’ the instantaneous graphical output of Simple Player™ application to create a snapshot of the application in operation.

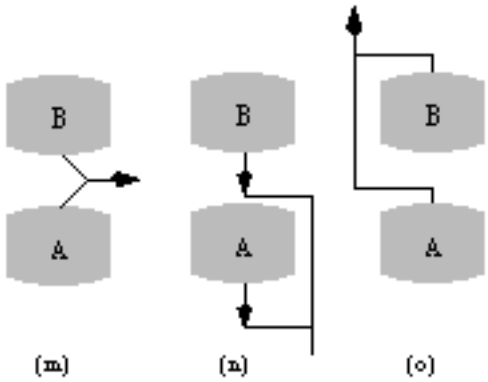


Figure 6.2. The types of connections which have been ruled out.

(d) Connection type (*aout, dinp*). Data is sent from the abstraction side of A to the display side of B. For example, the *thumb* interactor of the case study (chapter 5) sends data to the player bar and the selection band interactors.

(e) Connection type (*c, dinp*). The controller unit of interactor A and the input on the display side of interactor B are mutually constrained. This could be used to implement a mode, e.g. a keyboard modifier. Such an example can be found in the case study, where the interactor *xcontroller* applies constraints to mouse input to the display side of interactors *forward*, *backward* and *playPauseButton*.

(f) Connection type (*ainp, ainp*). The two interactors are synchronous consumers. Consider a graphical interface, which is resized following a menu command. One possible approach to modelling the interface is that interactors should receive the new screen coordinates of their enclosing window from the abstraction side.

(g) Connection type (*dout, ainp*). Data is sent from the display side of B to the abstraction side of A. This could be an example of a graphics output pipeline, where each interactor manages one transformation of the graphics data structures.

(h) Connection type (*aout, ainp*). A value is communicated from A to B. For example, in the case study the *resizeBox* interactor was connected to all interactors displayed with this type of connection.

(i-l) Connection types (*c, ainp*), (*c, dout*), (*c, aout*), and (*c,c*). In these cases the controller of interactor A constrains, and is constrained by, a gate of interactor B. This is similar to type (e) above. These types of connections were used broadly in the case study, e.g. for *xcontroller* and the *playPause* interactor.

The list above does not include all the possible connection types. Figure 6.2 illustrates those omitted. These connection types concern pairs of interactors which synchronise over common output gates. Clearly, this introduces the possibility of a deadlock when the two interactors output a different value. It is shown below how these connection types are excluded on the basis of the transformation requirement.

6.2.2 Correctness of the synthesis transformation for synchronous compositions

The distributed and the compound forms of the composition are:

$$DF = (ADU_A|[G^A_{io}]|CU_A)|[G]|(ADU_B|[G^B_{io}]|CU_B)$$
$$CF = (ADU_A|[G_1]|ADU_B)|[G^A_{io} \quad G^B_{io}]|(CU_A|[G_2]|CU_B) \qquad 6.13$$

Strong bisimulation equivalence of DF and CF is proved below. The proof results in constraints for the gate sets G_1 and G_2 so that DF~CF. The proof technique used here is adapted from [180] and is a 'shorthand' version of the proof by bisimulation. It is based on the following theorem by Milner [133]:

$$DF \sim CF \text{ iff } (DF,CF) \quad R \text{ and } R \text{ is a bisimulation relation.} \qquad 6.14$$

Because an ADC interactor specifies no 'silent' actions i, the following must hold for R to be a bisimulation relation [133].

For all interactions g<v> for which there is a transition for DF or CF:

$$DF \xrightarrow{g<v>} DF' \qquad CF'|(DF',CF') \quad R \cdot CF \xrightarrow{g<v>} CF'$$
$$CF \xrightarrow{g<v>} CF' \qquad DF'|(DF',CF') \quad R \cdot DF \xrightarrow{g<v>} DF' \qquad 6.15$$

Proof

The proof consists in proposing a relation R that can play the role of the bisimulation relation. For 6.15 to hold, necessary and sufficient conditions are derived relating the gate sets of the component processes of DF and CF.

$$\text{Let } R = \{<(A|[S_1]|B) \,|[S_2]|\, (C|[S_3]|D), (A|[S_4]|C) \,|[S_5]|\, (B|[S_6]|D)>\} \qquad 6.16$$

where A, B, C, and D are behaviour expressions, S_1 to S_6 are gate sets. Clearly, (DF,CF) R by substitution. From the semantics of the synchronisation operator, any transition of DF or CF, involves the transition of at least one of its components. Further, by the semantics of the synchronisation operator of LOTOS, the transition will always result in the *same* static structure for DF' and CF'. Thus, (DF',CF') R, for any transition g<v>.

Condition (6.15), requires that DF and CF have the same sets of transitions. A transition g<v> is defined by gate identifier g and value <v>. By the semantics of synchronous composition, the gate identifier g must belong to the gate set of the component process performing the transition. This gate set does not change with the recursive instantiation of the process. This is guaranteed by the convention adopted in paragraph 6.1.1 for the recursive instantiation of processes ADU and CU.

DF and CF are behaviour expressions formed by the same components. Therefore all possible transitions may be enumerated by considering each possible combination of transitions for their components. Transitions of DF and CF are categorised by the component processes involved. The domain of g for each type of transition is the intersection of the gate sets of the component processes, constrained by the behaviour expression DF or CF so that the transition is possible, according to the operational semantics of LOTOS [100, 18]. An empty gate set means that the transition is

ADU_A	CU_A	ADU_B	CU_B	Condition
•				impossible for both DF and CF
	•			$G_c^A \cap G = G_c^A \cap (G_2 \quad G_{io}^B)$
•	•			$G_{io}^A \cap G = G_{io}^A \cap (G_1 \quad G_2)$
		•		impossible for both DF and CF
•		•		impossible for both DF and CF
	•	•		$(G_{io}^B - G_1) \cap ((G_c^A \quad G_{io}^A) - G_2) =$
•	•	•		No resulting condition (=)
			•	$G_c^B \cap G = G_c^B \cap (G_2 \quad G_{io}^A)$
•			•	$(G_{io}^A - G_1) \cap ((G_{io}^B \quad G_c^B) - G_2) =$
	•		•	$G_c^A \cap G_c^B \cap G = G_c^A \cap G_c^B \cap G_2$
•	•		•	$G_{io}^A \cap G_c^B \cap G = (G_2 - G_1) \cap ((G_{io}^A \cap G_c^B) \quad (G_{io}^A \cap G_{io}^B))$
		•	•	$G_{io}^B \cap G = G_{io}^B \cap (G_1 \quad G_2)$
•		•	•	No resulting condition (=)
	•	•	•	$G_{io}^B \cap G_c^A \cap G = (G_2 - G_1) \cap ((G_{io}^B \cap G_c^A) \quad (G_{io}^A \cap G_{io}^B)$
•	•	•	•	$G_{io}^A \cap G_{io}^B \cap G = G_{io}^A \cap G_{io}^B \cap G_1 \cap G_2$

Table 6.1. Each row shows a combination of transitions and the condition put on the label sets of the components and G, G_1, G_2 for it to be possible for both DF and CF.

impossible. To ensure that DF and CF offer the same events, the domains for g of DF and CF are equated. Simple set manipulations result in the necessary conditions between the gate sets of the two expressions.

To cover all possible transitions for DF or CF, this procedure is repeated for all possible combinations of transitions of their components. The results are listed in table 6.1. Note that some combinations are impossible by the definition of the model, as for example an ADU firing without synchronising with the CU component that controls it. The rightmost column presents the condition resulting by equating the domain of g for the DF and the CF form, for each possible transition g<v>.

For example consider the first row of table 6.1. It describes the case of a transition of the form $ADU_A \rightarrow ADU'_A$. For the DF such a transition is impossible because by the definition of ADC in paragraph 6.1.8, all gates of the ADU synchronise with the gates of its corresponding CU, denoted below as L(CU).

The second row describes a transition of the form $CU_A \rightarrow CU'_A$. This is perfectly legitimate for both the DF and the CF. For DF the transition may happen through any of the gates of $L(CU_A)$, provided it does not belong to either the gates of ADU_A or the synchronisation gates in G. For the CF the transition may happen only if the gate g is

not in the synchronisation gates with CU_B, i.e. set G_2, or in the set of gates $G_{io}^A \quad G_{io}^B$, on which the two controllers synchronise with the two ADUs. By equating the two gate sets the condition listed on the table arises:

$$L(DF) = L(CF) \Leftrightarrow$$
$$G_c^A - G_{io}^A - G = G_c^A - G_2 - (G_{io}^A \quad G_{io}^B) \Leftrightarrow$$
$$G_c^A \cap G = G_c^A \cap (G_2 \quad G_{io}^B) \qquad 6.17$$

In this case a useful condition results. In others a tautology arises, (i.e. the condition Ø=Ø), which is noted as 'no condition' on the corresponding row of the table.

The combination of the resulting conditions allows many possible constructions for G_1 and G_2. The necessary and sufficient condition that results from their combination is

$$G_{io}^A \cap G_{io}^B \quad G, G_1, G_2 \qquad G_1 \cap ((G_{io}^A \cap G_C^B) \quad (G_{io}^B \cap G_C^A)) = \qquad 6.18$$

Condition 6.19 below is stronger than 6.18 (it is a sufficient condition) but it allows an economical construction of the compound form.

$$G_1 = G_{io}^A \cap G_{io}^B \cap G \qquad G_2 = G \qquad 6.19$$

Thus, the controllers in the CF must be composed over exactly the synchronisation gate set used in the DF, while the ADUs can not synchronise over the gates implementing connections of types (e), (i), (j), (k) and (l) of figure 6.1.

From the transformation requirement it is also required that G_1 does not include any common output gates. The transformation requirement 6.12 can only hold for the gate sets satisfying equation 6.18 if G itself does not include common output gates. Thus, for the transformation requirement to hold, the input form must be constrained to satisfy the following equation

$$G \cap G_o^A \cap G_o^B = \qquad 6.20$$

This, as was expected, rules out the connections of types (m), (n) and (o) of figure 6.2.

The discussion so far has ignored the data values passed. It is argued hereby that this does not discredit the proof presented. An interaction may involve the communication of data from one interactor to another, unless it involves both interactors reading data from a third source or simply synchronising. These two cases are examined separately.

Connections (c), (d), (g) and (h) are the only connection types that support data communication. Without loss of generality it is assumed that A produces data values and B consumes them. By the definition of the model, the gates of the ADU are typed. Let ValueSet(t) denote the possibly infinite set of values of sort t, and $range_A(g)$ ValueSet(t) (respectively $range_B(g)$ ValueSet(t)) denote the set of values possibly offered by ADU_A (respectively read by ADU_B) over gate g. As predicate selection was disallowed in the definition of the interactors it follows that: $range_B(g)$ = ValueSet(t).

Also, CU_A and CU_B, by definition, allow all values v ValueSet(t). In table 6.1, data communication takes place only in the transition in row (15). By inspection of the DF and the CF it is easy to see that in both cases the set of possible values is $range_A(g)$. Thus, for any gate label g, DF and CF offer the same transitions g<v>, with <v> $range_A(g)$.

The remaining types of connections, i.e. (a), (b), (e), (f), (i), (j), (k) and (l) specify the interaction of the two interactors with a variable declaration on both sides of a connection. In this case, both the DF and the CF offer the same set of interactions g<v> where g is the synchronisation gate, and <v> is any value in ValueSet(t).

Å

The above can be summarised in the form of a theorem.

Theorem 6.1. Synthesis of synchronous composition expressions.

A behaviour expression which specifies the synchronous composition of two ADC interactor specifications over a gate set G, such that $G \cap G_o^A \cap G_o^B =$, can be rewritten as a strongly equivalent ADC interactor. The CU of the resulting interactor is formed by the parallel composition of the two component CUs over the gate set G and its ADU is formed by the parallel composition of the component ADUs over a gate set

$$G_1 = G_{io}^A \cap G_{io}^B \cap G \qquad 6.21$$

where the gate sets for interactors ADC^A and ADC^B are indicated accordingly (figure 6.3). This rewriting, termed synthesis, preserves strong bisimulation equivalence, denoted by ~.

$$(ADU_A|[G_{io}^A]|CU_A)|[G]|(ADU_B|[G_{io}^B]|CU_B) \sim (ADU_A|[G_1]|ADU_B)|[G_{io}^A \quad G_{io}^B]|(CU_A|[G]|CU_B) \qquad 6.22$$

Å

The right hand side of the above equivalence is a behaviour expression that is an ADC interactor. This transformation is a form of distribution property of the synchronisation operator with the proviso that the ADUs synchronise not on the gate set G itself, but on a subset G_1 that excludes gates that reflect cross synchronisation of the ADU with the CU of the other interactors.

This theorem can be simplified for the case of interleaving where G = :

$$(ADU_A|[G_{io}^A]|CU_A)|||(ADU_B|[G_{io}^B]|CU_B) \sim (ADU_A|||ADU_B)|[G_{io}^A \quad G_{io}^B]|(CU_A|||CU_B) \qquad 6.23$$

In the case of full synchronisation, where $G = (G_{io}^A \quad G_c^A) \quad (G_{io}^B \quad G_c^B)$, and provided that $G \cap G_o^A \cap G_o^B =$, equation 6.22 may be simplified to the following form:

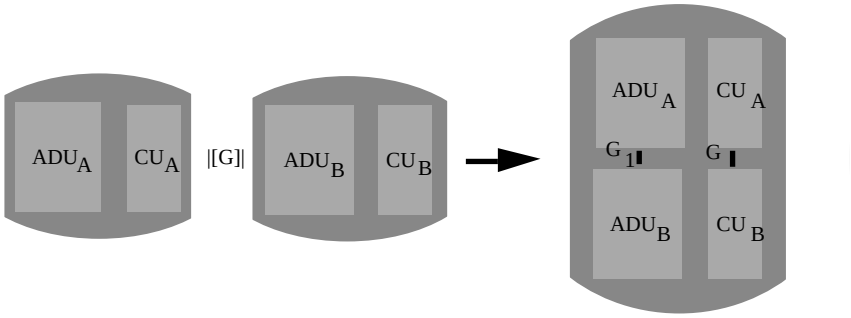


Figure 6.3. The synthesis of two interactors into one.

$$(ADU_A|[G^A_{io}]|CU_A)||(ADU_B|[G^B_{io}]|CU_B) \sim (ADU_A|[G^A_{io} \cap G^B_{io}]|ADU_B)|[G^A_{io} \quad G^B_{io}]|(CU_A||CU_B) \qquad 6.24$$

6.2.3 Correctness of the composition with [] (choice)

The same proof technique is used as with the synchronous composition of interactors. However, the proofs for choice and disable do not need to consider the communication of data between the interactors. The distributed and the compound forms of the transformation are as follows:

$$DF = (ADU_A|[G^A_{io}]|CU_A)[](ADU_B|[G^B_{io}]|CU_B)$$
$$CF = (ADU_A|||ADU_B)|[G^A_{io} \quad G^B_{io}]|(CU_A[]CU_B) \qquad 6.25$$

Choice can be used to specify alternative interactions. Consider, for example, a set of interactors for drawing different shapes on a drawing package. These may be invoked from an interactor that supports logical disjunction, e.g. a palette or a set of radio buttons. The set of alternative interactors could be related by the choice operator and their composition could be synchronised with the menu interactor on their start events. Unfortunately, the case study of chapter 5 has not given rise to examples of the use of choice and disable in interface specifications. For the sake of completeness, the correctness of the synthesis transformation for the operators choice and disable operators is discussed as well.

The bisimulation relation is defined to contain two pairs of structures, as follows:

$$R = \{a, b\} \text{ where}$$
$$a = < (A|[S_1]|B) [] (C|[S_2]|D), \ (A|||C) |[S_3]| (B [] D)> \qquad 6.26$$
$$b = <A |[S_1]|B, (A|||C) |[S_2]| B)>\}$$

where A-D are behaviour expressions, S_1-S_3 are gate sets. (DF, CF) is isomorphic to type (a) of elements of relation R, so (DF, CF) R. To satisfy the definition for the bisimulation relation DF and CF need to have exactly the same sets of transitions, and the result of the transition will also be a pair of behaviour expressions belonging to R. The structure of the behaviour expressions changes with the first transition into a pair

	ADU_A	CU_A	ADU_B	CU_B	Impossible for DF	Impossible for CF	Condition resulting from L(DF)=L(CF)
1	•				by ADC definition	by ADC definition	
2		•					$G_c^A \cap G_{io}^B =$
3	•	•					no condition
4			•		by ADC definition	by ADC definition	
5	•		•		by ADC definition	by ADC definition	
6		•	•		by ADC definition		$G_{io}^A \cap G_{io}^B =$ $G_c^A \cap G_{io}^B =$
7	•	•	•		by ADC definition	semantics of |||	
8				•			$G_c^B \cap G_{io}^A =$
9	•			•	by ADC definition		$G_{io}^A \cap G_{io}^B =$ $G_c^B \cap G_{io}^A =$
10		•		•	semantics of []	semantics of []	
11	•	•		•	semantics of []	semantics of []	
12			•	•			no condition
13	•		•	•	by ADC definition	semantics of |||	
14		•	•	•	semantics of []	semantics of []	
15	•	•	•	•	semantics of []	semantics of []	

Table 6.2. Transitions of pairs behaviour expressions (DF,CF) of type (a). Each row shows a combination of transitions of their components and the condition required from their label sets for the transition to be possible for both DF and CF for exactly the same events.

(DF',CF') of type b. Transitions for pairs of behaviour expressions (DF,CF) of type (b) maintain the same structure, so R indeed contains all the possible pairs of behaviour expressions that might result from a transition.

Both pairs of structures (a) and (b) refer to common components. The possible transitions are enumerated as all possible combinations of transitions of their components. The set of transitions have to be the same for the DF and CF, so the label sets that are involved in each set of transitions are equated. This is done for both pairs of structures in the relation R. For each pair of structures a table is constructed listing combinations of transitions for their components and the necessary conditions upon the gate sets.

Once the first transition has been made both DF and CF are reduced to a static composition expression. Consider the case where the choice is made in favour of CU_A. (The corresponding conditions for the case where a transition of CU_B occurs result symmetrically).

$$DF = ADU_A|[G^A_{io}]|CU_A$$
$$CF = (ADU_A|||ADU_B)|[G^A_{io}]|CU_A \quad 6.27$$

Table 6.3 does not introduce any conditions other than those already identified in table 6.2. (The same holds for the symmetrical case, where the choice has been resolved in favour of CU_B). In conclusion, the resulting necessary and sufficient conditions for DF~CF are as follows:

$$G^A_c \cap G^B_{io} = \qquad G^B_c \cap G^A_{io} = \qquad G^A_{io} \cap G^B_{io} = \quad 6.28$$

6.2.4 Correctness of the composition with [> (disable)

The disable operator is useful for describing interruption, e.g. evoked by a 'quit' button. The button itself can be specified as an interactor which a dialogue with the user to confirm that the application should be stopped. This dialogue could be specified as an interactor related with the rest of the interface with the disable operator.

The distributed and compound forms for the disable composition operator are:

$$DF = (ADU_A|[G^A_{io}]|CU_A)[>(ADU_B|[G^B_{io}]|CU_B)$$
$$CF = (ADU_A|||ADU_B)|[G^A_{io} \quad G^B_{io}]|(CU_A[>CU_B) \quad 6.29$$

The proof is almost identical to that presented above for the choice operator, so it is not repeated. The bisimulation relation is defined to be:

$$R = \{a, b\} \text{ where}$$
$$a = <(A|[S_1]|B)\ [>\ (C|[S_2]|D),\ \ A|||C\ |[S_3]|\ (B[>\ D)> \quad 6.30$$
$$b = \ <A\ |[S_1]|B,\ (A|||C)\ |[S_2]|\ B)>\}$$

The following necessary and sufficient condition results for the gate sets of the component processes:

$$G^A_c \cap G^B_{io} = \qquad G^B_c \cap G^A_{io} = \qquad G^A_{io} \cap G^B_{io} = \quad 6.31$$

The disable and choice operators, can easily be recognised as a useful structure in interface design specification. For example, disable may describe termination with 'quit' or 'close' dialogues, and choice might help specify alternative behaviours, e.g. different 'screens' evoked by the appropriate interaction. However, at present, the implications of using disable and choice to combine ADC interactor specifications has not yet been explored in practical examples. The last two results are summarised in the form of a theorem.

Theorem 6.2. Synthesis of dynamic composition expressions.

The composition of two ADC interactor specifications with a dynamic composition operator, i.e. choice [] or disable [>, can be rewritten as a strongly equivalent ADC

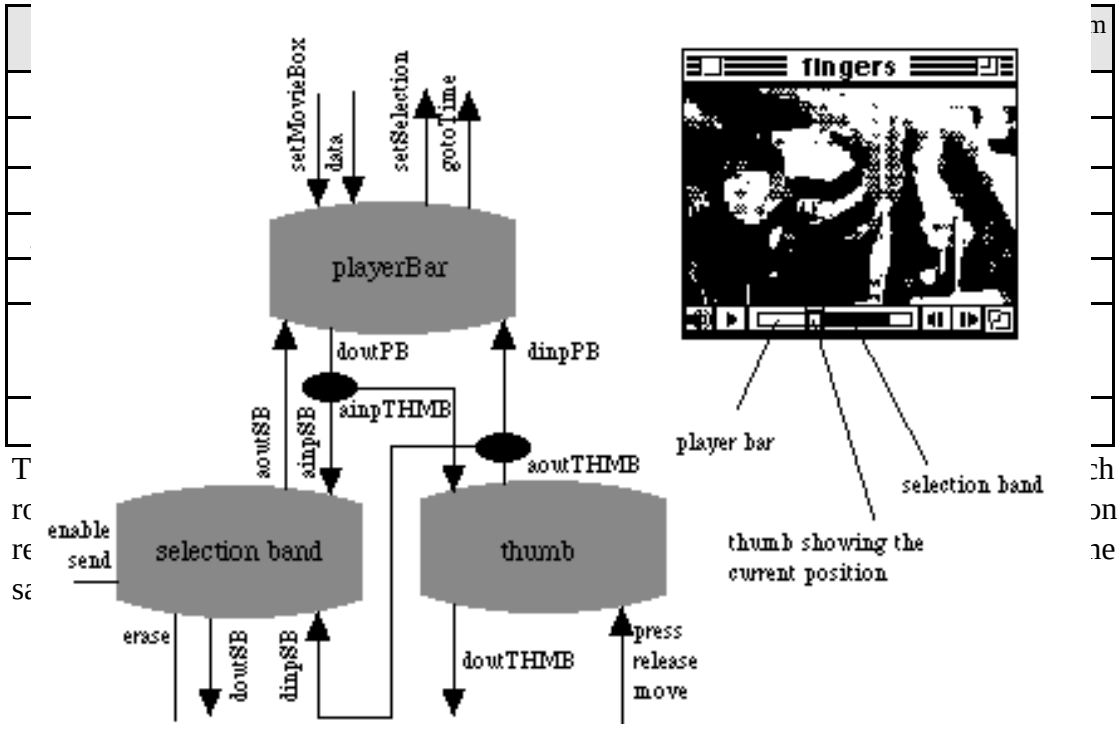


Figure 6.4. The composition of three interactors of Simple Player™.

interactor. The CU of this interactor is a behaviour expression which combines the two component CUs with the same dynamic operator. The ADU of the compound interactor is formed by the interleaved composition of the two component ADUs.

$$(ADU_A|[G^A_{io}]|CU_A) \quad (ADU_B|[G^B_{io}]|CU_B) \sim (ADU_A|||ADU_B)|[G^A_{io} \quad G^B_{io}]|(CU_A \quad CU_B) \qquad 6.32$$

where is either [] or [> and

$G^A_c \cap G^B_{io} =$ $G^B_c \cap G^A_{io} =$ $G^A_{io} \cap G^B_{io} =$

Å

6.2.5 Example

In the case study of chapter 5 the graphical interface to the Simple Player™ application for the Macintosh computer was modelled as a composition expression, whose operands are all elementary ADC interactors. A small segment of the graph which illustrated this composition expression, is examined here, to illustrate the application of the synthesis transformation. It is the group of interactors that handle all interaction with the player bar. This includes the interactor *playerBar*, the interactor *thumb* and the interactor *selectionBand*. Also, it includes two logical connectives necessary for the distribution of data among the interactors. The relevant segment of the graph of figure 5.5 is shown in figure 6.4.

In most user interface implementation architectures this group of interactors is implemented as a single unit. While the modular description was found easier for the purposes of the reverse engineering study, it is most likely that in a forward engineering design exercise these components are modelled as a single interactor. The synthesis transformation is applied upon the composition expression which corresponds to the graph segment of figure 6.4, and a single ADC interactor expression is obtained. In the context of the graph of figure 5.5 representing the global interface architecture, the graph segment can be replaced by a single barrel node representing the compound form.

Distributed Form (DF)

The top side of the *playerBar* interactor is connected to the functional core, not shown in figure 6.4. This core sends to the player bar the time coordinate of the currently shown frame of the movie played. The player bar interactor outputs its display state through gate *doutPB*. There are two recipients for this data. The thumb interactor receives the time information at gate *ainpTHMB* through the connective *dm1* shown as an ellipse, and the selection band receives the time information on gate *ainpSB*. The thumb interactor uses this information to update its display, offered as a display state at gate *doutTHMB*. Similar information can flow in the opposite direction. Mouse events are input to the interactor *thumb* from the display side, and communicated to the other two interactors from gate *aoutTHMB*, once more with the help of a connective.

For economy of presentation, where the gate set is obvious it is substituted by '...' and the following shorthand is used:

G_c= {start, restart, suspend, resume, abort}

The LOTOS behaviour expression defining the DF is:

```
((playerBar[Gc, dinpPB, aoutSB, doutPB, setMovieBox, data, data, gotoTime, setSelection]
    |[Gc, doutPB]|
dm1[Gc, doutPB, ainpTHMB, ainpSB])
    |[Gc, ainpTHMB, dinpPB]|
(thumb[Gc, press, move, release, doutTHMB, ainpTHMB, aoutTHMB]
    |[Gc, aoutTHMB]|
dm2[Gc, aoutTHMB, dinpPB, dinpSB]))
    |[Gc, dinpSB, aoutSB, ainpSB]|
selectionBand[Gc, dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]
```

For economy of space, it is only possible to show here a bit of the internal structure of one interactor. The interested reader is referred to [124], for a full exposition of the specification. Consider for example the *selectionBand* interactor. It is itself the composition of an ADU and a CU as follows:

```
process selectionBand[Gc, dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]:noexit:=
    aduSB[dinpSB, doutSB, ainpSB, aoutSB, erase](noSelection, thePlayerBar, thePlayerBar)
            |[dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]|
    cuSB [Gc, dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]
```

```
endproc
```

Process *aduSB* is initialised with an abstraction value *noSelection*, (defined in the data typing component of the specification), and an initial display state *thePlayerBar*, which is presumed to be the initial presentation of the player bar (with no selection band showing).

```
process aduSB[dinpSB, doutSB, ainpSB, aoutSB, erase] (a:line,pc,ps:playBar) : noexit :=
    aoutSB!a;                aduSB[...] (a,pc,ps) []
    doutSB!pc;               aduSB[...] (a,pc,pc) []
    ainpSB?x:playBar;        aduSB[...] (receivePB(a,x),renderPB(pc,x),ps) []
    dinpSB?x:pnt;            aduSB[...] (inputPnt(x,a),echo(x,ps,a),ps) []
    erase;                   aduSB[...] (inputEr(a),echoEr(pc, a),ps)
endproc
```

Process *cuSB* supports the standard control behaviours plus the particular to *selectionBand* dialogue constraints. For brevity only the constraints component *ccSB* of *cuSB* that describes them is shown.

```
process ccSB [dinpSB, doutSB, ainpSB, aoutSB, erase,  enable, send] : noexit :=
    (enable; (operation[dinpSB, doutSB, ainpSB, aoutSB, erase,  send]
    [> send; aoutSB?x:line; ccSB [...]))
[]  ainpSB?x:playBar;ccSB [...]
[]  doutSB?x:playBar; ccSB [...]
[]  erase; doutSB?x:playBar; aoutSB?x:line; ccSB [...]
endproc

process operation[dinpSB, doutSB, ainpSB, aoutSB, erase,  send] : noexit :=
    dinpSB?x:pnt; doutSB?x:playBar; operation[...]
[]  doutSB?x:playBar; operation[...]
endproc
```

Compound Form (CF)

Process *compound*, the CF, has the general ADC structure:

```
process compound[Gc, Gio] : noexit  :=
    adu[Gio] |[Gio]| cu[Gc,Gio]
endproc
```

where the following shorthand is used:

```
Gio = {setMovieBox, data, setSelection, gotoTime, doutPB, dinpPB, aoutSB, ainpSB, ainpTHMB,
            aoutTHMB, press, move, release, doutTHMB, dinpSB, doutSB, erase}
```

The ADU and the CU of the compound form are defined as follows:

```
process adu[Gio]: noexit :=
((aduPB[dinpPB, aoutSB, doutPB, setMovieBox, data, data, gotoTime, setSelection]
                    (initPlayBarData, thePlayerBar, thePlayerBar)
    |[doutPB]|
aduDM1[doutPB, ainpTHMB, ainpSB](thePlayerBar))
    |[ ainpTHMB, dinpPB]|
```

```
(aduTHMB[ press, move, release, doutTHMB, ainpTHMB, aoutTHMB](indicator, thumb, thumb)
    |[ aoutTHMB]|
aduDM2[aoutTHMB, dinpPB, dinpSB](aPoint)))
    |[dinpSB, aoutSB, ainpSB]|
aduSB[dinpSB, doutSB, ainpSB, aoutSB, erase](noSelection, thePlayerBar, thePlayerBar)
endproc

process cu[Gc,Gio]: noexit :=
((cuPB[Gc, dinpPB, aoutSB, doutPB, setMovieBox, data, data, gotoTime, setSelection]
    |[Gc, doutPB]|
cuDM1[Gc,doutPB, ainpTHMB, ainpSB])
    |[Gc, ainpTHMB, dinpPB]|
(cuTHMB[Gc, press, move, release, doutTHMB, ainpTHMB, aoutTHMB]
    |[Gc, aoutTHMB]|
cuDM2[Gc, aoutTHMB, dinpPB, dinpSB]))
    |[Gc, dinpSB, aoutSB, ainpSB]|
cuSB[Gc, dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]
endproc
```

6.2.6 Discussion

A reflection on the outcome of the synthesis transformation gives rise to some questions. The compound form is not simpler than the distributed form and it does not seem to enhance modularity, at least in any obvious way. The controller units have been 'factored out', and they have been 'divorced' from their corresponding ADUs. The synthesis transformation is simply a re-writing of the behaviour composition expression and not some simplification of it. Some such simplifications are examined in later sections, where common behaviours are factored out and simpler expressions result. In that context, synthesis enables the regrouping of the component processes, which is necessary prior to any simplification of the behaviour expression.

Rather than simplifying the specification text, the synthesis transformation changes the 'grain' of the architectural description. For example, it may be better to think of the player bar of this example as a single interactor (the compound form) and not as a group of interactors. Whether or not this is the case depends on the purpose and the context of the specification exercise and the designed architecture. The regrouping of interactors supported by the synthesis transformation does not change the semantics or the complexity of the specification. Rather, it moulds the specification into the form that fits the architectural design of the interface. For example, at the initial stages of an interface design, a single compound interactor describing the interface to Simple Player™ may be a more appropriate description, rather than the distributed form corresponding to the graph of figure 5.5.

If a higher level description is adopted, many of the details of the specification and the configuration of the interactors will be irrelevant. This may be the case for a bottom-up design process, where the compound form is used as a building block for more complex expressions without reference to its internal detail. It is equally the case for top-down

design where specifications of pre-defined components are combined, or where the configuration of the interactors has not yet been worked out. In such cases, the need for a more abstract description of interactors is apparent and this abstraction is the subject of the next section.

Synthesis does not apply generally to LOTOS specifications; the proof of correctness is contingent upon the specific form of the ADC process definition. Regarding the synchronous composition of interactors, synthesis is effectively a re-shuffling of the operands and synchronisation gates in a parallel composition of processes. A similar re-shuffling was introduced as a congruence law for Basic LOTOS processes in [180], but the conditions required there do not hold for the ADC model. The reshuffling of a synchronous composition can be seen as a special case of the graphical composition theorem of [19], summarised in appendix A.2. The maximal cooperation condition which is required by the graphical composition theorem is stronger than the conditions of table 6.1. In particular, 6.22 holds by the graphical composition theorem applies only where $G = (G_{io}^{A} \quad G_{c}^{A}) \cap (G_{io}^{B} \quad G_{c}^{B})$ and where $G_1 = G_{io}^{A} \cap G_{io}^{B}$ and $G_2 = G$. Finally, the proof presented here is more general as it extends to the disable and choice operators as well.

Relevant work reported by Faconti et al. in [64] is concerned with the synchronous composition of logical input devices for graphics systems to form hierarchies (their approach could easily apply to the interactor model of Paternó and Faconti [150]). The equivalence of all possible textual specifications of a diagrammatic representation is established on the basis of the graphical composition theorem. This validates the graphical representation as a more accurate and precise model of the specification and motivates the support of a visual editor. In the terms of the previous sections the equivalence discussed is between a set of distributed forms. In contrast, the approach presented here focuses on the synthesis of a compound form that maintains the general ADC structure.

6.3 Abstract Views of Interactors

The *hide* operator of LOTOS was introduced in section 3.8.8. A *hide* expression designates that parts of some specified behaviour can not be observed by its environment. The *hi*de operator takes as an argument a set of gates which are hidden. Hiding may apply to an interactor or to a group of interactors in order to abstract from internal detail. The term *abstract view* is adopted here to describe an interactor some of whose gates are hidden. In the figures abstract views are illustrated as black frames enclosing the interactors-barrels.

Abstract views can be used as building blocks for interface specifications. As was the case with interactor specifications, it is interesting to examine in what circumstances a composition expression whose arguments are abstract views can itself be reshaped into an abstract view. This section shows that abstract views of interactors maintain, in large, the compositionality characteristics of interactors. Synthesis applies to abstract views as well. Below, theorems 6.1 and 6.2 for the synthesis of interactors are extended to describe the synthesis of abstract views.

Theorem 6.3. Synchronous composition of abstract views.

Consider the abstract views of two interactors ADC_A and ADC_B which hide gates G_h^A and G_h^B respectively and which synchronise at gates:

$$G \cap G_o^A \cap G_o^B = \qquad 6.33$$

where the gate sets of the interactors are indicated accordingly. This synchronous composition expression can be rewritten as the abstract view of an ADC interactor provided that any one abstract view does not hide gates of the other. The CU of the resulting form is described by the parallel composition of the two component CUs over the gate set G. The ADU of this form is the parallel composition of the two component ADUs over the gate set:

$$G_1 = G_{io}^A \cap G_{io}^B \cap G \qquad 6.34$$

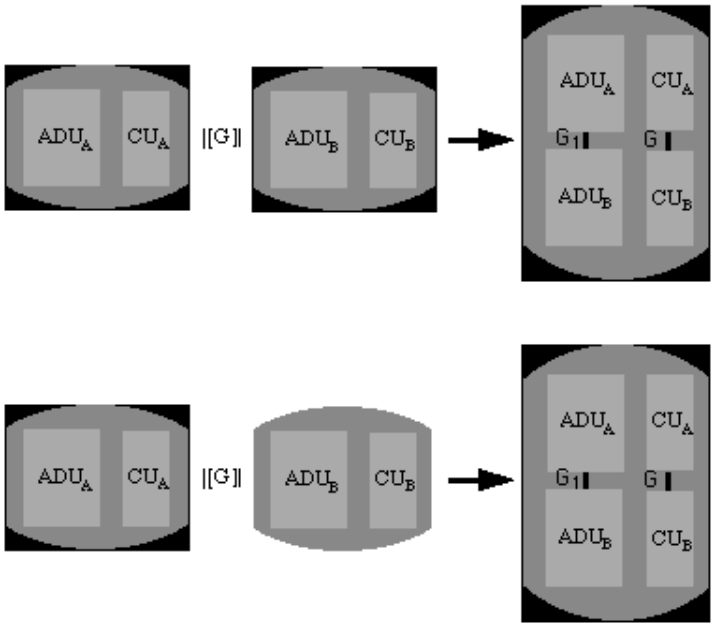


Figure 6.5. Synthesis applied (a) to the synchronous composition of two abstract views, and (b) to the composition of an abstract view and an interactor.

The resulting abstract view hides the union of the two sets of gates hidden by the component abstract views, so

$$G_h^{AB} = G_h^A \quad G_h^B \qquad 6.35$$

Under the condition mentioned above synthesis preserves weak observational congruence, denoted by .

$$\text{If } G_h^A \cap (G_{io}^B \quad G_c^B) = \quad \text{and } G_h^B \cap (G_{io}^A \quad G_c^A) = \quad \text{then}$$
$$\text{hide } G_h^A \text{ in } (ADU_A|[G_{io}^A]|CU_A)|[G]| \text{ hide } G_h^B \text{ in } (ADU_B|[G_{io}^B]|CU_B) \qquad 6.36$$
$$\text{hide } G_h^{AB} \text{ in } (ADU_A|[G_1]|ADU_B)|[G_{io}^A \quad G_{io}^B]|(CU_A|[G]|CU_B)$$

Proof

It is known [100, pp.90] that the following weak observational congruence holds for any two LOTOS behaviour expressions B_1 and B_2.

$$\text{hide A in } B_1|[A']|B_2 \quad (\text{hide A in } B_1)\ |[A']|(\text{hide A in } B_2) \text{ if } A \cap A' = \qquad 6.37$$

Strong observational equivalence entails weak observational congruence (cf. [133]), the required congruence 6.36 follows by 6.37 and the theorem 6.1.

Å

In the special case where $G_h^B \cap (G_c^B \quad G_{io}^B) =$, equation 6.37 describes the composition of an abstract view with an interactor (see figure 6.5.b). If the same holds for ADC_A then 6.37 reduces to the equivalence of theorem 6.1.

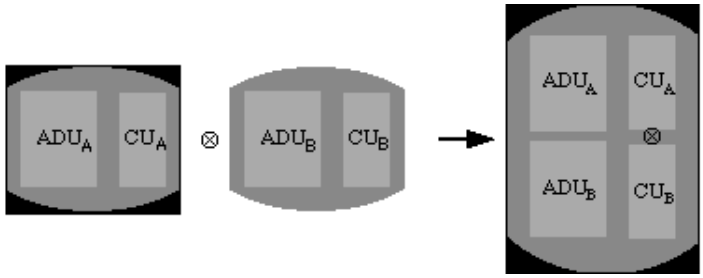


Figure 6.6. Synthesis of dynamic compositions of abstract views.

Theorem 6.4. Dynamic Composition of Abstract Views.

Consider the abstract views of two interactor specifications ADC_A and ADC_B which hide gates G_h^A and G_h^B respectively. Their composition by choice [] or disable [> can be rewritten as an abstract view which hides the union of G_h^A and G_h^B. The CU of the compound form combines the two component CUs with [] or [> respectively. The ADU of the compound form is formed by the interleaved composition of the two component ADUs. This rewriting preserves weak observational congruence:

$$\begin{array}{l}\text{hide } G_h^A \text{ in } (ADU_A|[G_{io}^A]|CU_A) \quad \text{hide } G_h^B \text{ in } (ADU_B|[G_{io}^B]|CU_B) \\ \quad \text{hide } G_h^B \quad G_h^A \text{ in } (ADU_A|||ADU_B)|[G_{io}^A \quad G_{io}^B]|(CU_A \quad CU_B)\end{array} \qquad 6.38$$

where is [] or [> and $G_c^A \cap G_{io}^B =$ $G_c^B \cap G_{io}^A =$ $G_{io}^A \cap G_{io}^B =$

This rewriting is illustrated in figure 6.6.

Proof

It is known [100, pp. 90] that hiding distributes over dynamic composition operators. Thus the following weak observational congruences hold for any two LOTOS behaviour expressions B_1 and B_2.

hide A in B_1[]B_2 = (hide A in B_1) [] (hide A in B_2)

hide A in B_1[>B_2 = (hide A in B_1) [> (hide A in B_2)

Let B_1= hide G_h^A in ADC_A and B_2= hide G_h^B in ADC_B. By substitution of 6.32 the congruence 6.38 follows.

Å

Another congruence law for hiding ([100, pp. 90]) states that successive applications of *hide* can be 'factored out' of a behaviour expression:

$$\text{hide A in (hide A' in B)} \quad \text{hide A} \quad \text{A' in B} \qquad 6.39$$

It is easy to see that the cumulative effect of applying synthesis successively to a behaviour expression which involves abstract views and interactors is to rewrite the expression as the abstract view of a compound ADC interactor:

$$\text{hide [...all hidden gates...] } (ADU_{\text{top level}} |[G_{\text{top level}}]| CU_{\text{top level}}) \qquad 6.40$$

where the ADU_{top_level} is a well formed ADU and the CU_{top_level} is isomorphic to the original behaviour expression. This answers some question raised earlier in this section. Abstract views support compositionality provided that the hidden gates of one abstract view and the gate set of the other processes do not overlap. This is not a severe constraint in writing specifications, but it does mean that some caution should be exercised in naming the gates of an interactor. This problem can be overcome easily by associating actual gate names with an identifier for the interactor they belong to.

The congruences shown in this section followed, in a straight forward manner, from the corresponding equivalences defined for ADC interactors. They are interesting to this thesis, because they show that the notion of an abstract view is an essential extension to that of an interactor. As far as the synthesis transformation is concerned, abstract views can be manipulated in the same manner as interactors. In the congruences discussed, rewriting behaviour expressions involving ADC interactors result in expressions involving abstract views. This suggests that abstract views are themselves essential 'building blocks' for complex specifications.

6.4 Decomposition

Theorems 6.1 to 6.4 summarise the synthesis transformation, each describing the equivalence of two behaviour expressions. Synthesis applies to the left hand side of each equivalence, the distributed form, and regroups its components so as to produce the right hand side, the compound form. When read from right to left, these equivalences describe the inverse of the synthesis transformation. This is called here the *decomposition* of interactors.

The decomposition transformation corresponds to a very general and intuitive design step where one software element is substituted with the composition of two others. Decomposition transformation is defined here so that its input expression has the form that results from the synthesis transformation. The problem of shaping any ADC interactor specification to this required form is discussed in more detail below.

1. The input expression for the decomposition transformation is:

$$CF = ADU_{AB}|[G_{io}^{AB}]|CU_{AB} \qquad 6.41$$

where

$ADU_{AB} = ADU_A|[G_1]|ADU_B$ and $CU_{AB} = CU_A \quad CU_B$
if is $|[G]|$ with $G \cap G_o^A \cap G_o^B =$ then $G_1 = G_{io}^A \cap G_{io}^B \cap G$
if is [], [> then $G_1 =$

The naming of the components implies that they should be well-formed ADUs and well-formed CUs respectively.

2. The output expression for the decomposition transformation is:

$$DF = (ADU_A|[G_{io}^A]|CU_A) \quad (ADU_B|[G_{io}^B]|CU_B) \qquad 6.42$$

3. The transformation requirement is that the resulting component processes are themselves ADC interactors. This is true trivially by the definition of the ADC interactor, of section 6.1.8, and because the input form requires that the components appearing in 6.41 are well formed.

4. The correctness preservation requirement is that the distributed form is weak observational congruent to the compound form. This requirement holds on the basis of theorems 6.1 and 6.2.

This transformation is simply a regrouping of the LOTOS processes that comprise the compound form. This form is quite restrictive so decomposition, as defined here, is not as widely applicable as the synthesis transformation. In general, an ADC interactor does not have the form described by 6.41, and to mould it into this form the ADU and the CU have to be decomposed first. This problem is discussed in the remainder of this section.

6.4.1 Decomposition of an elementary ADU

Consider an elementary ADU that manages multiple abstraction and display sorts of the same or different data type AD. It may be a useful transformation to decompose this ADU into two or more elementary ADU components, each managing a single pair of abstraction and display sorts, or possibly one of the two only. The decomposition of this special form of ADU is discussed below.

The definition of the ADU relates every one of its gates to a pair of abstraction and display sorts. For example a *dout* gate will output a display sort, an *ainp* gate will apply a *receive* operation to update some abstraction sort and a *render* operation to update the display sort. For the argument that follows it makes no difference that some interactors can be abstraction-only or display-only. Clearly, the state of the ADU is observed from the output gates and is defined by the value of the state parameters. The correspondence of gates to abstraction and display sorts partitions the sets of gates of the ADU. Each of the subsets of its gates can only observe and effect behaviour relevant to these sorts. In such a case it is possible to decompose the ADU with a simple transformation to obtain two or more components which partition its gate sets.

In the following, instantiations of the ADU are subscribed, e.g. ADU_ϑ. An action of the ADU is denoted as g<v>, where g is a gate and <v> stands for some action specification. A transition of the ADU is described as using the following system of indexing:

$$ADU_\vartheta \xrightarrow{g<v>} ADU_{\vartheta\, gv} \qquad 6.43$$

Definition. Restriction operator.

For the purposes of this discussion, it is useful to introduce a restriction operator \G similar to that of CCS [133, chapter 2]. Restriction is a unary postfix operator whose argument G is a gate set. It applies to a behaviour expression and it prohibits any interaction on the gate set G. Its operational semantics can be described as:

$$\text{if } g \in G \text{ and } P \xrightarrow{g<v>} P' \text{ then } P\backslash G \xrightarrow{g<v>} P'\backslash G \qquad 6.44$$

Consider a partitioning of gates G_{io} of an elementary ADU to two gate sets G_A and G_B, such that each manages distinct state parameters. Using the restriction operator this condition can be described as follows:

$$\begin{aligned} &\text{If } G_{io} = G_A \cup G_B \text{ and } G_A \cap G_B = \varnothing \text{ then} \\ &\quad g \in G_A | ADU_\vartheta \xrightarrow{g<v>} ADU_{\vartheta\, gv} \cdot ADU_{\vartheta gv} \backslash G_B \sim ADU_\vartheta \backslash G_B \\ &\quad g \in G_B | ADU_\vartheta \xrightarrow{g<v>} ADU_{\vartheta\, gv} \cdot ADU_{\vartheta gv} \backslash G_A \sim ADU_\vartheta \backslash G_A \end{aligned} \qquad 6.45$$

Equivalence 6.45 means that when the ADU has multiple state parameters, the effect of operations applied to one sort (respectively, a pair of sorts) cannot be observed from the gates that correspond to the other sorts.

A transformation T is defined that transforms an ADU which satisfies 6.45 to an observationally equivalent behaviour expression $ADU_A|||ADU_B$ with respective gate sets G_A and G_B. The definition of T follows a similar technique to that used in [28]. Transformation T is defined as follows.

$$\begin{aligned} &T(ADU) = ADU_A \,|||\, ADU_B \\ &\text{where } ADU_A = T_A(ADU) \text{ and } ADU_B = T_B(ADU) \end{aligned} \qquad 6.46$$

The mappings T_A and T_B are defined in a compositional way for all operations that may be involved in the definition of an elementary ADU.

1. Action Prefix

$$\begin{aligned} &g \in G_A | B = g<v>;B_{gv} \cdot T_A(B) = g<v>;T_A(B_{gv}) \quad T_B(B) = stop \\ &g \in G_B | B = g<v>;B_{gv} \cdot T_A(B) = stop \quad T_B(B) = g<v>;T_B(B_{gv}) \end{aligned} \qquad 6.47$$

2. Choice

$$T_A(B_A[]B_B) = T_A(B_A)[]T_A(B_B) \text{ and } T_B(B_A[]B_B) = T_B(B_A)[]T_B(B_B) \qquad 6.48$$

3. Process instantiation P

$T_A(P)=P_A$ and $T_B(P)=P_B$ 6.49

4. Process definition

If P:=B then $P_A:=T_A(B)$ and $P_B:=T_B(B)$ 6.50

The correctness preservation requirement for T is

ADU~T(ADU) 6.51

The proof of 6.51 is facilitated by first proving the following lemma.

Lemma. Disjunction of complementary components of an ADU.

An ADU which satisfies 6.45 can be described in the constraint oriented style as the disjunction of two independent behaviours. Using the restriction operator of 6.44, each of these behaviours is described as the restriction of the ADU to a subset of its gates. These gate sets are defined to satisfy 6.45, i.e. they provide access to different state parameters.

$ADU \sim ADU\backslash G_A \,|||\, ADU\backslash G_B$ 6.52

Proof of the lemma.

ADU is recursive so 6.52 can be shown by the method of transition induction, i.e. by induction on the length of the transition. It will be shown that

$(ADU, ADU\backslash G_A ||| ADU\backslash G_B)$ S where S is a bisimulation relation. 6.53

Clearly both expressions have the same sets of transitions g<v>. In particular the following can be stated

g G_A, ϑ a state of ADU and < v > an action specification:

$ADU_\vartheta \xrightarrow{g<v>} ADU_{\vartheta gv}$ iff 6.54

$ADU_\vartheta \backslash G_A ||| ADU_\vartheta \backslash G_B \xrightarrow{g<v>} ADU_{\vartheta gv} \backslash G_A ||| ADU_\vartheta \backslash G_B$

On the basis of 6.45 the last transition is rewritten as:

$ADU_\vartheta \backslash G_A \,|||\, ADU_\vartheta \backslash G_B \xrightarrow{g<v>} ADU_{\vartheta gv} \backslash G_A \,|||\, ADU_{\vartheta gv} \backslash G_B$ 6.55

From a shorter length of induction it can be assumed that:

$(ADU_{\vartheta gv}, ADU_{\vartheta gv}\backslash G_A ||| ADU_{\vartheta gv}\backslash G_B)$ S 6.56

The symmetric argument can be made for all g G_B. Therefore

$$(ADU_\vartheta, ADU_\vartheta\backslash G_A ||| ADU_\vartheta\backslash G_B) \quad S \tag{6.57}$$

The state ϑ of the ADU has not been constrained so it could be the initial state. It follows that S satisfies the definition of a bisimulation relation, and so 6.52 is true.

û

Proof. Correctness of the transformation T.

The shorthand $\sum$ is introduced to denote choice from an indexed set of interactions. An elementary ADU has the form

$$ADU_\vartheta = \sum_{g_i\ G_A\ G_B} g_i; ADU_{\vartheta g_i v} \tag{6.58}$$

T is applied to this expression.

$$T(ADU_\vartheta) = ADU_{A\vartheta} ||| ADU_{B\vartheta}$$

where

$$ADU_{x\vartheta} = T_x\left(\sum_{g_i\ G_A\ G_B} g_i < v >; ADU_{\vartheta g_i v}\right), \quad x \quad \{A,B\} \tag{6.59}$$

$$T_x(ADU_\vartheta) = \sum_{g_i\ G_A} g_i < v >; T_x(ADU_{\vartheta g_i v})$$

On the basis of the lemma 6.52 it is sufficient to show that

$$ADU_\vartheta\backslash G_A \sim ADU_{A\vartheta} \text{ and } ADU_\vartheta\backslash G_B \sim ADU_{B\vartheta} \tag{6.60}$$

This is shown trivially by showing that $(ADU_\vartheta\backslash G_A, ADU_{A\vartheta})$ is a bisimulation relation, and correspondingly for $(ADU_\vartheta\backslash G_B, ADU_{B\vartheta})$.

û

6.4.2 Decomposition of a well formed ADU

A well formed ADU can be a parallel composition of simpler ADU components. The transformation T is extended to cope with a well formed ADU by the addition of a fifth rule:

5. If $B = B_1|[G]|B_2$ then

$$T_A(B)=T_A(B_1)\ |[G\text{-}B]|\ T_A(B_2) \text{ and } T_B(B)=T_B(B_1)\ |[G\text{-}A]|\ T_B(B_2) \tag{6.61}$$

Proof Correctness of T.

The correctness of the transformation can be shown by structural induction. The transformation has been proven correct for the elementary ADU components. The induction hypothesis is that

$$B_1 \sim T_A(B_1) \,|||\, T_B(B_1) \text{ and } B_2 \sim T_A(B_2) \,|||\, T_B(B_2) \qquad 6.62$$

By substitution it follows that

$$B \sim (T_A(B_1) \,|||\, T_B(B_1)) \,|[G]|\, (T_A(B_2) \,|||\, T_B(B_2)) \qquad 6.63$$

The right hand side of equation 6.63 satisfies the maximal cooperation condition, so by the graphical composition theorem (appendix A.2) equation 6.63 can be rewritten as :

$$B \sim (T_A(B_1) \,|[G\text{-}B]|\, T_A(B_2)) \,|||\, (T_B(B_1) \,|[G\text{-}A]|\, T_B(B_2)) \qquad 6.64$$

which proves the correctness of 6.61.

û

The transformation T can be used to decompose a well formed ADU or an elementary ADU that manages multiple state parameters. An example of its application is discussed in section 7.3. Transformation T does not help decompose an elementary ADU which manages one abstraction and/or one display sort only, as in the examples of chapter 4 and 5.

At this point it is interesting to compare the proposed transformation with related transformations that apply to standard LOTOS processes. In general the problem of decomposition has been solved for finite non-recursive processes only. For example, [118] reports the decomposition of basic LOTOS finite processes into two processes that partition its gate sets, only for the case where the process is written in an action prefix form (a normal form using the terminology of [133]). An improvement has been proposed in [28] that works for a wider range of basic LOTOS expressions. However, the restrictions imposed prohibit recursive processes and multiple instantiations of the same process, so a different transformation needed to be defined for the ADU.

So far in this chapter the internal structure of the CU has not been detailed, nor have there been any special behavioural requirements set upon it. It can be any LOTOS process. The data component of an action specification is irrelevant to the workings of the CU. It is only necessary in order to synchronise with the ADU. At this stage the decomposition of the CU is an instance of the general problem of decomposing a basic LOTOS process. The algorithms of [28] and [118, chapter 2] or the use of the inverse expansion of [118, chapter 3] are possible partial solutions. In the case where a CU is simply a CC, decomposition is more applicable, particularly when the CC is written in a constraint oriented style.

6.5 Parameterised behaviours

The ADC interactor definition, of section 6.1, does not stipulate any syntactic or behavioural requirements for the CU. The synthesis and decomposition transformations introduced so far support this abstract definition of the ADC model. This section details

the definition of the CU, to model the standard behaviours that have been labelled collectively SSRRA in chapter 4. They are specified constructively by adopting a standard structure for the CU and declaratively by referring to properties of the labelled transition system that models the interactor behaviour.

In the process definitions below, the keywords *process*, *endproc*, and *where*, and the functionality specification of each process (*exit* or *noexit*) are omitted for brevity. The definition below is a generalisation of the one in section 4.7.

```
CU[{start, suspend, resume, restart, abort}   Gd   Gio] :=
    start; RUN[{suspend, resume, restart, abort}   Gd   Gio]

RUN[{suspend, resume, restart, abort}   Gd   Gio] :=
    (CC[Gd   Gio] |[Gd   Gio]| SU_RE[{suspend, resume}   Gd   Gio])
[>  INT[{suspend, resume, restart, abort}   Gd   Gio]

SU_RE[{suspend, resume}   Gd   Gio]:=
    ANY[Gd   Gio] [>suspend; resume; SU_RE[{suspend, resume}   Gd   Gio]

ANY[G]= choice g in G [] g<sort(g)>; ANY[G]

INT[{suspend, resume, restart, abort}   Gd   Gio] :=
    restart; RUN[{suspend, resume, restart, abort}   Gd   Gio]
[]  abort; stop
```

Where the shorthand <sort(g)> is the empty string when the gate g is used for simple synchronisation and ?x:s for a gate associated with a sort s.

The SSRRA behaviours may be described in terms of the set of traces of the ADC process Tr(ADC).

1. The interactor starts with an event on gate start

$$\sigma \quad Tr(ADC) \cdot first(\sigma) = \{start\} \qquad 6.65$$

2. An interactor may be suspended with an event on gate *suspend*. After suspension the interactor does not offer any interactions of the constraints component, i.e. on gates G_{io} G_d. Once suspended the interactor can only resume with an event on a gate resume. It resumes at exactly the same state it was before the suspension.

$$\begin{aligned} &\sigma \quad Tr(ADC)|last(\sigma) = suspend \cdot out(ADC) = \{resume, restart, abort\} \\ &\sigma \quad Tr(ADC)|last(\sigma) \neq suspend \cdot ADC_{\sigma;suspend.resume} \sim ADC_{\sigma} \end{aligned} \qquad 6.66$$

3. At any moment in the operation of the interactor a restart event will have the effect of returning the interactor to the dialogue state following the start operation.

$$\sigma \quad Tr(ADC) \cdot ADC_{start} \sim_{\phi} ADC_{\sigma;restart} \qquad 6.67$$

Where ϕ is a bijective coding function which relates LOTOS actions with the same gate identifier (see appendix A.1). This means that a restart event will result in a

state which is φ-bisimilar to the state of the interactor after its initialisation. 6.66 compares the temporal behaviour of the two processes and not the values of the state parameters. The state parameters are not re-initialised with a restart event. Such an operation could be encoded in the AD data type and not the CU component.

4. An abort event terminates the interactor behaviour.

$$\sigma \quad Tr(ADC)|last(\sigma) = abort \cdot ADC_{\sigma} \sim stop \qquad 6.68$$

6.5.1 Synthesis and the SSRRA behaviours

In general the synthesis transformation does not preserve the structure of the CU. The CU of the resulting compound form is a LOTOS expression whose operands are the CUs of the participating interactors. Below, a special case is discussed, where it is possible to rewrite this expression into the form of the CU. The component processes and SSRRA gate identifiers are annotated to indicate the process they correspond to. Also, G_x will denote G^x_{io} G^x_d, and $SSRRA_x$= {$start_x$, $suspend_x$, $resume_x$, $restart_x$, $abort_x$}. The general CU structure is now written as:

CU_x = $start_x$; RUN_x
RUN_x= (CC_x|[G_x]|SU_RE_x)[> INT_x
SU_RE_x= ANY_x [> $suspend_x$; $resume_x$;SU_RE_x
ANY_x= choice g in G_x [] g; ANY_x
INT_x= $restart_x$; RUN_x [] $abort_x$; exit

Consider the synchronisation of CU_A and CU_B which synchronise over their SSRRA gates (where $SSRRA_A$=$SSRRA_B$). This was the case with all the interactors of the case study.

CU_{AB} = CU_A |[G]| CU_B where SSRRA G.

The controller CU_{AB} of the compound form can then be written in the general CU structure above as follows:

CU_{AB} = start; RUN_{AB}
RUN_{AB}= (CC_{AB}|[G_{AB}]|SU_RE_{AB}) [> INT_{AB}
SU_RE_{AB}= ANY_{AB}[> suspend; resume; SU_RE_{AB}
ANY_{AB}= choice g in G_{AB} [] g; ANY_{AB}
INT_{AB}= restart; RUN_{AB} [] abort; exit
where CC_{AB}= CC_A |[G-SSRRA]| CC_B

This construction may be used for the example of section 6.2.5. The CU of the compound form is a parallel composition expression of several CU components. Their common elements can be factored out, and the constraints component would be the following

process cc[...]: noexit :=
((ccPB[dinpPB, aoutSB, doutPB, setMovieBox, data, data, gotoTime, setSelection]

```
    |[doutPB]|
ccDM1[doutPB, ainpTHMB, ainpSB])
    |[ainpTHMB, dinpPB]|
(ccTHMB[press, move, release, doutTHMB, ainpTHMB, aoutTHMB]
    |[aoutTHMB]|
ccDM2[aoutTHMB, dinpPB, dinpSB]))
    |[dinpSB, aoutSB, ainpSB]|
ccSB[dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]
endproc
```

The choice and disable operators cannot be factored into the CC component in a similar fashion as above. In an expression of the form $CU_A[]CU_B$ (or $CU_A[>CU_B$) the first interaction commits the compound CU to either operand (respectively an interaction of CU_B will commit the interactor to the second operand). This is also true when they have identical start gates, in which case, the choice is made non-deterministically at the first interaction of the compound interactor.

In conclusion, the standard structure for the SSRRA cannot be maintained through the synthesis transformation apart from the special case when the interactors synchronise on their SSRRA gates. This case is not so uncommon. In the case study of chapter 5 this was the case for all the interactors specified. Common SSRRA behaviours characterise self contained segments of interaction, e.g. dialogue boxes, one page of a spreadsheet application, or one screen on a hypertext document. Navigation between such components, their invocation or say the dynamic creation of file icons by a file manager are characterised by distinct SSRRA behaviours. In such cases the compound CU cannot be simplified further than is suggested by the synthesis transformation, in theorems 6.1-6.4. This observation shows that the synchronisation over the SSRRA gates can be used to specify some sort of grouping. For example, interactors associated with closely related tasks or functions can be grouped together to reflect the structure of the tasks they support.

6.5.2 Decomposition of the CU

This paragraph investigates the preservation of the CU structure through decomposition. Decomposition, as defined in section 6.4, assumes that the CU of the compound form is a pair-wise composition of two CU components. In the opposite case, the CU will have to be shaped into this form. This paragraph examines the special case where the CU is written in the constraint oriented style [180]. This refers to the specification of the CC, the only component of CU with a variable specification style.

The CC component describes the ‘custom’ dialogue properties of the interactor modelled. In the constraint oriented style of specification, discussed in section 3.8, each logical constraint on the dialogue of the interactor is expressed as a behaviour expression. These expressions are composed by means of parallel operators. Interleaving combines the logical constraints in a disjunction. Synchronised composition specifies the conjunction of the logical constraints expressed.

Theorem 6.5. Decomposition of a constraint oriented CU.

Consider a CU which has the standard structure of paragraph 6.5.1 and whose CC is written in the constraint oriented style as the parallel composition of constraints CC_A and CC_B. These processes describe temporal ordering constraints on gate sets A and B respectively. They synchronise on a set of gates G which must be a subset of $A \cap B$.

$$CC_{A\ B} = CC_A \ |[G]|\ CC_B \text{ where } G \quad A \cap B \qquad 6.69$$

The CU can be rewritten in the strongly equivalent form

$$CU_A \ |[SSRRA \quad G]|\ CU_B \qquad 6.70$$

where CU_x is defined as in paragraph 6.5.1.

Proof

This theorem is shown in a series of steps. For brevity the gates *start, suspend, resume, restart, abort* are denoted as *st, su, re, rs, ab*.

1. By the definition of SU_RE

$$SU\_RE_{A\ B} = ANY_{A\ B}[> su; re; SU\_RE_{A\ B} \qquad 6.71$$

by expansion it is easy to see that if G $A \cap B$ then

$$SU\_RE_{A\ B} \sim SU\_RE_A|[\{su, re\} \quad G]|SU\_RE_B \qquad 6.72$$

2. By bisimulation or the graphical composition theorem of [19] (appendix A.2) it may be shown that:

$$((CC_A|[A]|SU\_RE_A)|[\{su,re\} \quad G]|(CC_B|[B]|SU\_RE_B)) \sim$$
$$(CC_A|[G]|CC_B) \ |[A \quad B]|(SU\_RE_A|[\{su,re\} \quad G]|SU\_RE_B) \qquad 6.73$$

3. From 6.72 and 6.73 and 6.69 it follows that:

$$(CC_A|[A]|SU\_RE_A)|[\{su, re\} \quad G]|\ (CC_B|[B]|SU\_RE_B) \sim$$
$$CC_{A\ B}|[A \quad B]|SU\_RE_{A\ B} \qquad 6.74$$

4. Lemma. Distribution of restart and abort behaviours.

Let P, Q be processes with gate sets $G_P \cap \{rs, ab\} =$ and $G_Q \cap \{rs, ab\} =$.

Also let

$$T_P = P[>(rs; T_P \ [] \ ab;stop)$$
$$T_Q = Q[>(rs; T_Q \ [] \ ab;stop) \text{ and} \qquad 6.75$$
$$T_{PQ} = (P|[G_s]|Q) \ [> (rs; T_{PQ} \ [] \ ab;stop)$$

It can be shown by bisimulation that:

$T_P|[\{rs, ab\} \quad G_s]|T_Q \sim T_{PQ}$ 6.76

5. Lemma 6.76 is applied for $P=(CC_A|[A]|SU\_RE_A)$ and $Q=(CC_B|[B]|SU\_RE_B)$, and $G_s=G \quad \{su, re\}$. By substitution:

$T_P=(CC_A|[A]|SU\_RE_A)[> (rs; T_P [] ab;stop)$

$T_Q=(CC_B|[B]|SU\_RE_B)[> (rs; T_Q [] ab;stop)$ 6.77

$T_{PQ}=(CC_A|[A]|SU\_RE_A)|[G_s]|(CC_B|[B]|SU\_RE_B)[>(rs; T_{PQ} [] ab;stop)$

By 6.74 the last process definition can be rewritten as

$T_{PQ}=(CC_{A\ B}|[A \quad B]|SU\_RE_{A\ B})\ [> (rs; T_{PQ} [] ab;stop)$ 6.78

6. An interaction on gate *st* can be prefixed to both sides of 6.76, preserving strong equivalence:

$st; (T_P|[G \quad SSRRA]|T_Q) \sim st; T_{PQ}$ 6.79

and by the definition of CU this equivalence is rewritten as

$(CU_A|[G \quad SSRRA]|CU_B) \sim CU_{A\ B}$ 6.80

û

Corollary.

Consider an ADC interactor with gates $G_d \quad G_{io}$ SSRRA, such that it is possible to solve the decomposition problem for its ADU so that

$$G_{io}^A = A \cap G_{io},\ G_{io}^A = B \cap G_{io} \text{ and } G \quad A \cap B \cdot$$
$$ADU[G_{io}] = ADU_A[G_{io}^A]|[G]|ADU_B[G_{io}^B] \qquad 6.81$$

and further, its CC is written in the constraint based style:

$CC[A \quad B]:= CC_A[A]\ |[G]|\ CC_B[B]$ 6.82

Then it is possible to decompose the interactor specification to two interactors, which synchronise at least on the SSRRA gates. The composition expression relating the ADC interactors is as follows:

$$ADC = (ADC_A|[G \quad SSRRA]|ADC_B) \text{ where}$$
$$ADC_A = ADU_A|[A]|CU_A \text{ and } ADC_B = ADU_B|[B]|CU_B \qquad 6.83$$

The decomposition described by 6.83 is an ideal case where there are no temporal constraints across the two gate sets A and B. In general, this will not be the case, and these constraints over a set of gates AB A B will be described by a process $CC_{AB}[AB]$. Then the constraints component is written in the constraint oriented style as follows:

$$CC[A\ B]:= CC_A[A] \,|[G]|\, CC_B[B] \,|[AB]|\, CC_{AB}[AB] \qquad 6.84$$

Theorem 6.5 can be applied twice for the CU, once with the gate sets A B and AB A B, giving

$$CU_{A\ B}=CU_{(A\ B)\text{-}AB} \,|[SSRRA\ \ AB]|\, CU_{AB} \qquad 6.85$$

and subsequently for gate sets A and B to split the controller unit $CU_{(A\ B)\text{-}AB}$ and the ADU_{AB}. The final result would be a decomposition as below:

$$\begin{aligned} &ADC = (ADC_A|[G]|ADC_B)|[AB]|CU_{AB} \\ &ADC_A = ADU_A|[A]|CU_A \text{ and } ADC_B = ADU_B|[B]|CU_B \end{aligned} \qquad 6.86$$

This result also demonstrates the use of a CU as stand alone component, for capturing temporal constraints across the gates of two different interactors. Note that this decomposition results in a cluster of interactors which synchronise on their SSRRA gates, supporting the notion of a group as described earlier in this section.

6.6 Dialogue Modelling

In the context of human computer interaction the term dialogue is conventionally interpreted as the syntactic structure of the interaction between user and computer. The notion of syntax is rooted in linguistic models [77, 78] of interaction that distinguish the lexical, syntactic, and semantic levels of abstraction. In chapter 2 the dialogue component of the Arch reference model was discussed. This component is responsible for task-level sequencing, both for the user and for the portion of the functional core sequencing that depends upon the user. It has been mentioned already that the scope of the dialogue component is not at all clear cut, a fact demonstrated by the Slinky analogue, discussed in section 2.3.

In the framework of the ADC model dialogue is, nominally, the temporal ordering of interactions on its gates. It is stressed that what is modelled as dialogue in the ADC model is not inherently a property of either the specificand, i.e. the modelled interface, or the ADC model itself. What one specifier chooses to encode as dialogue, another may choose to specify in the semantics of data operations. This choice is similar to the choice between specification styles for operations on data, e.g. as discussed in [76], or the choice between specification notations as discussed in [144].

The barrier between the dialogue component and the presentation component is also fuzzy. In the case of the ADC model there is no distinct lexical level. It is defined

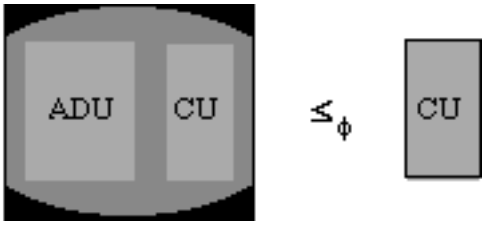


Figure 6.7. The dialogue of an abstract view can be observed sufficiently by an abstract view of the CU component alone.

implicitly by the lowest level of abstraction of the events modelled. For example, if the mouse input is considered as the lowest-level input action, then this is where the barrier is put, and similarly if a higher level of input interactions is considered, e.g. a menu selection this will define a different barrier. The same holds for output actions.

Within the framework of the ADC model, dialogue design for a single interactor consists partly in the instantiation of the standard (SSRRA) behaviours and partly in designing the temporal ordering for the remaining interactions. This latter set of behaviours is defined entirely in the constraints component (CC) of the CU.

From the understanding of what is dialogue it follows that the data values associated with events may be ignored when discussing dialogue properties. This may be achieved in practice by a transformation from Full LOTOS to Basic LOTOS, e.g. the naive transformation supported by the Lite toolset [30, 119] and reported in [118]. The intuition of what is dialogue can now be rephrased as below.

Definition. Dialogue representation in ADC.

The dialogue of an ADC specification is described by P(ADC), where P(.) is the naive transformation from a full LOTOS specification to a basic LOTOS specification defined over the same set of gates, and which maps each specified interaction g<v> to an interaction g.

From the definition of the ADC model and the definition of the transformation P(.), (see appendix A.1, or [118]), it follows easily that:

$$P(ADC) \sim P(CU) \qquad 6.87$$

Therefore the dialogue of the interactor is sufficiently modelled by P(CU).

By definition the naive transformation P(.) distributes over the hide operation. Thus, the dialogue of an abstract view is given by a hiding applied to the transformation to basic LOTOS of the controller unit (figure 6.7).

$$P(\text{hide g in } (ADU \;|[G_{io}]|\; CU)) \approx \text{hide g in } P(CU) \qquad 6.88$$

The equivalences 6.87 and 6.88 are very useful for the verification of dialogue properties of the interface design specification. Dialogue verification concerns a basic LOTOS

specification only. This is useful to overcome the practical difficulty in verifying properties of full LOTOS specifications and also of managing complex behaviour expressions.

The dialogue for an interface specified as the composition of many ADC interactors is distributed in the interactor specifications, but also, it is described by the behaviour expression that combines these interactors. To verify the dialogue of a composition expression involving many interactors or abstract views, it is sufficient to define their synthesis and to verify the dialogue on a basic LOTOS expression involving the CU of the compound interactor. This expression is obtained easily. Theorems 6.1-6.4 have shown that the CU of the compound form which results from the synthesis of interactors or abstract views is isomorphic to the input behaviour expression.

6.7 Conclusions

This chapter has discussed the ADC model and operations upon ADC interactor specifications in considerable depth and rigour. Variations of the basic ADC model were discussed, e.g. with composite ADU components, interactors that handle many sorts of data, or abstract views. The synthesis transformation and the decomposition transformation were defined. Synthesis merges two interactors or abstract views into one. Decomposition achieves the reverse. It is quite a common aspiration to perform such correctness preserving transformations of a specification, so related work for LOTOS specifications has been cited regularly. That the transformations result also in ADC interactors or abstract views, is an essential property of the formal model. The concept of an ADC interactor applies both to the interface as a whole and to its components. The transformations establish a link between these two views. The preservation of the structure of the ADC model has dictated the agenda for this investigation, and it is this structure that enabled solutions to these transformations to be identified, whereas they do not exist in general for LOTOS processes.

Chapter 6 has been a considerable formalisation exercise so here, as a recapitulation, the results of this exercise are presented in an informal and illustrative manner.

The formal definition of an interactor required the rigorous definition of its constituent components. Figure 6.8 shows the legitimate components for interactors: well formed ADUs and CUs. The CU may have the concrete structure that supports the parameterised the SSRRA behaviours or may act simply as a constraints component (CC).

Figure 6.8. Components of ADC interactors.

The synthesis and the decomposition transformations apply to a host of objects, that may be interactors, abstract views, or controller units only. The range of the possible entities to be used is indicated in figure 6.9. They include ‘degenerate’ interactors which are display-only and abstraction-only, which are

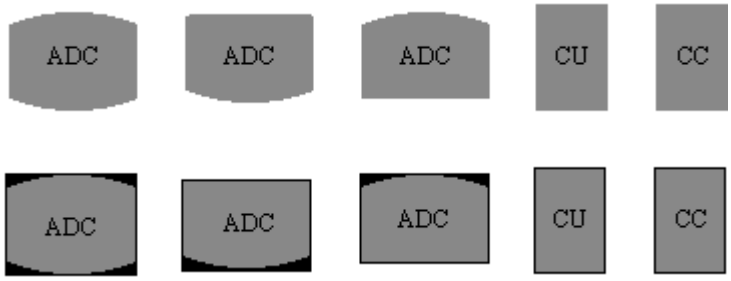


Figure 6.9. Illustrations of the range of possible interactors and abstract views.

indicated by barrels sliced along their axis. The rectangles representing controller units and constraints components are where the dialogue design for a particular interactor is encoded.

Hiding is important in scaling-up the use of interactors, in that it provides an abstract view of an interactor composition expression, and so allows the use of a compound interactor as a building block without considering its internal behaviour. It has been demonstrated that abstract views of interactors can be composed with interactors and other abstract views, provided gate identifiers do not collide, resulting in an abstract view of the composition. This shows that abstract views are themselves essential design constructs.

Synthesis is straight forward; it applies to all the objects in this domain provided the gate lists satisfy the conditions stipulated by theorems 6.1-6.4. If one of the objects synthesised is an abstract view, the result will also be an abstract view. The barrels shown in figure 6.10 stand for any of the types of interactors shown in figure 6.9, and similarly the ones in black frames could correspond to any of the types of abstract views in figure 6.9.

Decomposition is not as straight forward. It can be broken down to two stages. In the middle column of figure 6.11, interactors and abstract views are assumed to have the internal structure of the compound form that results from a synthesis transformation. If

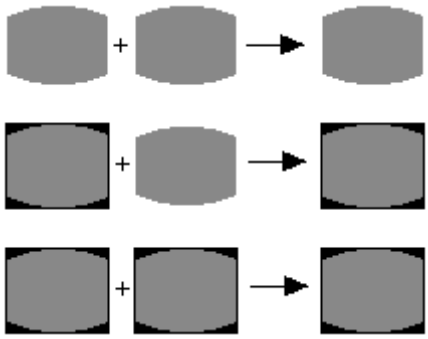


Figure 6.10. The synthesis transformation of well formed interactors or their abstract views is always possible.

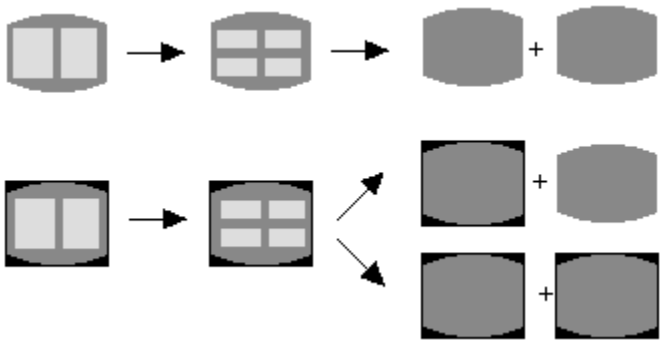


Figure 6.11. The decomposition is possible only if it is possible to bring the interactor or the abstract view to have the required form.

the compound interactor has this internal organisation the decomposition to a form described by the third column is always possible. The first column contains objects with the general form of an interactor or abstract view, i.e. any of the objects shown below the line in figure 6.9. The transition from this column to the second pertains to the general problem of decomposition of LOTOS processes. This is not always solvable. In section 6.4, some partial solutions for the decomposition of the ADU component were discussed, an algorithm was proposed and its correctness was proven, .

One of the aspirations of the model, discussed in chapter 4, was to model generic re-usable behaviours. A more concrete version of the ADC interactor model includes the parameterised description of behaviours that may be common across many interactors. It was shown that, in general, this extension to the model is detrimental for its modularity. A few exceptions were pointed out, where this extended structure of the ADC interactor is preserved through synthesis and decomposition. The most interesting is where the interactors are composed in parallel. Interactors related in this way may form part of the same coherent logical group, starting together, suspending, and interrupting synchronously. This can be a dialogue box, a panel of push buttons, etc. Another example is the Simple Player interface studied in chapter 5. This difficulty of preserving the semantics of the parameterised behaviours during synthesis and decomposition indicates some limits to specifying an interactor model by what is essentially a syntactic extension to the language. The result is a trade-off between the facility of using the parameterised definitions of SSRRA behaviours and the compositionality of the specifications.

Chapter 7

Use of the ADC model

This chapter discusses some applications of the ADC interactor model. The first two sections concern the analytical use of the model, for the specification and the verification of properties related to the usability of a specified user interface. Section 7.1 examines a class of properties pertaining to the relation of the abstraction and the display of an interface. The formal specification of this class of properties has been one of the original aims of the ADC model. The proposed formalisation improves on an earlier publication [123]. Section 7.2 discusses the specification and verification of dialogue properties. This discussion integrates, in the framework of the ADC interactor model, research results originating from different models. Sections 7.3 and 7.4 demonstrate the use of the ADC model within two design approaches. The first, stepwise refinement, is rooted in the traditions of formal methods in software engineering. The second, task based design, is more psychologically founded, using models of users' knowledge of their tasks to inform the design of a user interface. Section 7.4 discusses a simple formal description of a task model, originally proposed in [128]. This model is related to an interface specification, to formalise essential intuitions for task based design, extending a brief discussion on the subject in [122]. An overriding concern throughout this chapter is how workable these applications of the ADC model are in practice, how they are served by current tool support, and, perhaps more important, how verification results are interpreted, particularly in making predictions concerning the users.

7.1 Predictability and observability properties

The class of properties discussed hereby are related to the usability of a user interface. Originally, these properties were discussed as Generative User Engineering Principles (GUEPS) [85, 170]. As mentioned in chapter 3, abstract models of interaction [49 ,84] were introduced in an attempt to formalise these properties, with minimal commitment to a particular implementation architecture. Sufrin and He [169] developed a constructive model for the specification and classification of these properties. This scheme was used also by Abowd [1] to describe these properties systematically, in the framework of his

Agent model. It is adopted here also, to model these properties in terms of the ADC interactor model, in the formal framework of Labelled Transition Systems (LTS). LTS have been introduced briefly in section 3.7.

Predictability properties describe what can be inferred about the future behaviour of an interactive computer system from its current state. Their formal expression relies on the comparison of the instantaneous 'state' of the specified system to its subsequent behaviour. *Observability* properties pertain to what inferences can be made regarding the 'internal' behaviour of an interactive computer system from what can be observed externally from its display. Their formal expression relies on the definition of distinct 'views' of the system, one reflecting its internal workings and another reflecting its observation from the display.

The notion of a 'state' seems intuitive and straight forward, but it varies considerably across the formal models discussed so far. Abstract models of interaction assume a set of states, e.g.[49, 84], which are not described or enumerated. Models with more structure, like the Agent model of Abowd [1], describe states as a mapping of attributes to values. In the process algebraic framework of LOTOS, a state of a process corresponds to a state of the underlying Labelled Transition System. This state may be described by a behaviour expression that specifies the future behaviour of the process. Comparing states amounts to comparing behaviour expressions, e.g. they may be strong observational equivalent, weak observational equivalent, etc.

The formalisation of predictability and observability in the framework of abstract models of interaction is based upon the concepts of *equivalence* and *indistinguishability* of states, e.g. [49, 84]. Equivalence describes an apparent and temporary similarity of states at a particular instance during the interaction. For example, if at two distinct moments during interaction the display is the same, then the corresponding states of the interface are called display equivalent. Indistinguishability pertains to the relation between two states that cannot be shown to differ by subsequent interaction, i.e. when they are subjected to the same interactions. Unfortunately, this terminology is confusing as, in the context of process algebrae, it is the latter notion that is associated with the term 'equivalence'. To overcome this collision of terms, the designations '*similar'* and '*equivalent'* are introduced with the definitions below. Respectively, they correspond to the relations of 'equivalence' and 'indistinguishability' for the abstract state based models.

Definition. Status of a process.

The *status* of a process, at a given moment in time, is described by the interactions it can participate in. Let P denote a behaviour expression. P may offer to its environment a set of interactions, denoted as out(P). To specify only the interactions offered on a set of gates G, $out_G(P)$ is defined as:

$$out_G(P) = \{g \quad G | P \quad ^{g}\} \quad \{g < v > | g \quad G \quad v \quad valueSet(g) \quad P \quad ^{g<v>}\} \qquad 7.1$$

where valueSet(g) denotes the set of possible values that may be associated with interactions on gate g. For a variable declaration g?x:s, where s is some sort identifier, valueSet(g) includes all possible values of this sort. Otherwise, it may contain a single value that is output on gate g, by some value declaration g!v.

Definition. Abstraction-similar and Display-similar interactors.

The state-display model [84] introduced the distinction between the result and the display components of an interface model. This distinction has been built into the ADC model. Indeed, it has been one of the motivations for its definition. An interactor may be associated with multiple *result* operations. For each gate g G_{aout} a single event *g!result(A)* is offered, where A is the abstraction held by the interactor. Similarly the interactor outputs its display state D on the display side as: *dout!D*.

Two interactors P and Q (or two states of the LTS which represents an interactor) are called *abstraction-similar* (*display-similar*) when they are defined with identical gate sets G_{aout} (respectively G_{dout},) and when they output the same values on those. To exclude the contrived case where no events are offered on a gate, the clause is added that these sets should not be empty.

$$\begin{aligned} &similar_A(P,Q) \text{ iff } out_{G_{aout}}(P) = out_{G_{aout}}(Q) \neq \\ &similar_D(P,Q) \text{ iff } out_{G_{dout}}(P) = out_{G_{dout}}(Q) \neq \end{aligned} \qquad 7.2$$

Definition. Abstraction-side and Display-side behaviours of an interactor.

Interactors are characterised by the behaviour they exhibit when observed from the abstraction or the display sides. The *abstraction-side* behaviour of the interactor is observed by interaction at gates G_{aout} and G_{ainp}. The *display-side* behaviour is observed at gates G_{dinp} and G_{dout}. They are described by the pseudo-LOTOS expressions:

$$\begin{aligned} A_P &= \text{hide } (G_c \quad G_{dinp} \quad G_{dout}) \text{ in } P \\ D_P &= \text{hide } (G_c \quad G_{ainp} \quad G_{aout}) \text{ in } P \end{aligned} \qquad 7.3$$

Definition. Abstraction and Display Equivalence.

Two interactors P and Q are called *abstraction equivalent* if their abstraction-side behaviours are observationally equivalent, i.e. $A_P \approx A_Q$. They will be called *display equivalent* when their display-side behaviours are observationally equivalent, i.e. $D_P \approx D_Q$.

The meaning of these definitions depends on the equivalence relation ≈ they prescribe. There have been different proposals as to how to define the observable behaviour of a process and when two such behaviours should be deemed to be equivalent, e.g. [45, 133]. The choice between such proposals is a contentious issue which, in the present context, impinges on the ability of humans to tell apart interactive behaviours and to detect and interpret differences of the display contents. The relevant discussion is deferred until section 7.4, which questions the psychological validity of such definitions. Weak observational equivalence [133] is used in this section. Weak observational

1	Display Predictability	$similar_D(P,Q) \quad D_P \approx D_Q$
2	Result Predictability	$similar_A(P,Q) \quad A_P \approx A_Q$
3	Honesty (static)	$similar_D(P,Q) \quad similar_A(P,Q)$
4	Trustworthiness(static)	$D_P \approx D_Q \quad similar_A(P,Q)$
5	Strong WYSIWYG (static)	$similar_D(P,Q) \quad A_P \approx A_Q$
6	Weak WYSIWYG (static)	$D_p \approx D_Q \quad A_P \approx A_Q$
7	Honesty (dynamic)	$similar_A(P,Q) \quad similar_D(P,Q)$
8	Trustworthiness (dynamic)	$similar_A(P,Q) \quad D_P \approx D_Q$
9	Strong WYSIWYG(dynamic)	$A_p \approx A_Q \quad similar_D(P,Q)$
10	Weak WYSIWYG(dynamic)	$A_p \approx A_Q \quad D_P \approx D_Q$

Table 7.1. Expressions of predictability and observability properties.

equivalence distinguishes processes only with respect to observable interactions with their environment (see definition in appendix A.1), so it is a more realistic notion than strong observational equivalence [133]. Weak observational equivalence is an attractive proposition for this section, as its verification is supported by model checking tools, e.g. [68].

Using these definitions, a summative classification of observability and predictability properties similar to [1, pp.160] can now be written as in table 7.1. Consider, for example, the first row of the table. An interactor is called *display predictable*, if the similarity of two instances of its display implies that they are also display equivalent. In other words, the display status of the interactor determines its display-side behaviour. A symmetrical definition of *result predictability* can be written as in row 2 of table 7.1.

Observability properties, are described in rows 3-10 of table 7.1. For example, if the current output on the display side determines the current result, then the interactor is called *honest*. The user may infer that differences, and therefore changes, in the result will be observable as differences in the display. A weaker requirement is that the display side behaviour determine the current result. In this case the interactor is called *trustworthy*. Differences in the result status of the interactor, are possible to detect at the display side, by further interaction.

The famous What You See Is What You Get (WYSIWYG) property can be formalised in a weak and strong form. Row 5 of table 7.1 describes strong WYSIWYG. A similarity in the display of two interactor states implies abstraction equivalence. This is paraphrased as “What you see now determines what you will be able to get”. Row 6 of table 7.1 describes weak WYSIWYG: “All you can possibly see determines what you

will be able to get". In terms of the ADC model, this means that display equivalence implies abstraction equivalence.

The last four rows of table 7.1 correspond to what Abowd [1] called result oriented interaction. They have not been discussed as GUEPS or as desirable properties of interaction, but the classification scheme of [1], which is also adopted here, suggests their plausibility. At a first glance, these four cases describe to what extent the abstraction behaviour of the interactor determines the display behaviour. In most graphical interfaces the display contains more detail than the abstraction [172]. As Dix argues [49, chapter 7], if an interactive system is seen as the composition of a 'display layer' and an 'inner system layer', there should be a surjective mapping from states and operations of the display layer to those of the inner system layer. This view of interactive systems does not seem consistent with a result-oriented interaction, and so this may be the reason why previous formalisations of GUEPS, in [1] and [169], have concentrated on the dark shaded rows of table 7.1.

A closer examination reveals that the expressions of rows 7 to 10 of the table, are complementary to the four expressions listed above them. For example, consider row 7 of table 7.1, where abstraction-similarity implies display-similarity. This means that a change of the display informs the user of a change of the abstraction state. This property is called dynamic honesty, to show that it discloses changes in the interactor abstraction state.

For example, consider an electronic messaging application that operates in the background, while the user is engaged in unrelated tasks. The user is alerted to the arrival of a new message by an icon being superimposed on the display. Consider also, a user who infers that the state of the messaging application has changed from the change of the display. When the display has not changed, i.e. there is no icon on the screen, the user infers that a message has not arrived. According to the definition of honesty, of row 3, an honest system warrants the second inference but not the first. A system that displays a message-arrival icon without reason is still statically honest. An interface that satisfies the dynamic honesty requirement will correct this problem. On the other, hand a system that does not display the icon when a message arrives is dynamically honest but not statically honest. Clearly, true honesty should require both conditions to hold.

A weaker condition than dynamic honesty is *dynamic trustworthiness*. When result similarity implies display similarity, it is possible to infer changes in the result through experimentation with the display. The corresponding property in Sufrin and He's model was called "goal determines view". The dynamic strong WYSIWYG formulation means that it is possible to detect differences in the abstraction-side behaviour from instantaneous changes in the display side. In other words, a non similarity on the display side portrays a non equivalence of the abstraction side behaviours. The formal expression of the weak WYSIWYG stipulates that it should be possible to detect differences in the abstraction-side behaviour from differences in the display-side behaviours, i.e. a non equivalence on the display side portrays a non similarity of abstraction side behaviours.

Honesty	$similar_D(P,Q)$ $similar_A(P,Q)$ $\neg similar_D(P,Q)$ $\neg similar_A(P,Q)$
Trustworthiness	$(D_P \approx D_Q \quad similar_A(P,Q))$ $(D_P \not\approx D_Q \quad \neg similar_A(P,Q))$
Strong WYSIWYG	$(similar_D(P,Q) \quad A_P \approx A_Q)$ $(\neg similar_D(P,Q) \quad A_P \not\approx A_Q)$
Weak WYSIWYG	$(D_p \approx D_Q \quad A_P \approx A_Q)$ $(D_p \not\approx D_Q \quad A_P \not\approx A_Q)$

Table 7.2. The conjunction of the static and dynamic expressions of predictability and observability properties.

For a system to be called honest, trustworthy, etc. the conjunction of the static and dynamic versions of the properties has to be satisfied. The shaded rows of table 7.1 are merged, in table 7.2. Clearly, these expressions could be written more concisely using logical equivalence ($\Leftrightarrow$), but the implications of table 7.2 illustrate more clearly the inferences the user is justified to make by observing the display.

From the user's perspective, it is, perhaps, more interesting to predict the effect of a command or a sequence of commands, rather than to assert that the system is display predictable or result predictable. This property is defined in a manner similar to the definition of Sufrin and He [169]. A sequence of user commands *s* is said to be *directly predictable* if it has the same effect on the result in situations which are display similar. It is *predictable* if it has the same effect on the result in situations which are display equivalent. In table 7.3, if P is an interactor state and *s* is a sequence of user input, then let $P \quad {}^{s}P^{s}$.

Directly Predictable	$similar_D(P,Q) \quad A_P^s \approx A_Q^s$
Predictable	$D_P \approx D_Q \quad A_P^s \approx A_Q^s$

Table 7.3. Predictability of a sequence of user-input s.

The formalisations of tables 7.1-7.3 appeal to the same intuitions as the corresponding definitions by Sufrin and He [169], Abowd [1] and Dix [49]. Their interpretation differs slightly and this is attributed to the differences of the respective models. The ADC interactor is a communication entity that supports the users' interaction with the functional core of an interactive system, rather than encapsulating it. In the ADC model it is only the events communicated by the interactor on its two sides that are examined. Thus, the similarity predicate was defined in terms of the output gates from which the status of the abstraction and the display status are observed. The behaviour on either side is observed at both input and output gates.

Unfortunately, the definitions of predictability of tables 7.1-7.3 are too strong. For example, a closer look shows that no reasonable ADC interactor can be display predictable. By the definition of the model, the display does not change until an output event takes place. Therefore, immediately before and immediately after any input action on the display side, similarity is maintained:

$$P \quad S \mid g < v > \quad out_{dinp}(P) \quad P \xrightarrow{\quad} Q \cdot \; similar_D(P,Q) \qquad 7.4$$

However, the display behaviour after such an interaction does normally change, as a result of the input action. For the interactor to be predictable, the display side behaviour should never change:

$$P \quad S \mid g < v > \quad out_{dinp}(P) \quad P \xrightarrow{\quad} Q \cdot \; D_P \approx D_Q \qquad 7.5$$

Clearly this cannot be true for any useful interaction, as this would mean that user interactions do not influence the behaviour of the system as it is observed from the display.

This problem arises because an event based formalism is used to model status, a concept that is better described by a state based formalism. In fact, in the LOTOS specification both the display and the abstraction are modelled by state parameters. Similarity is a comparison of two statuses. The predictability related properties describe what may be inferred from a status regarding the behaviour.

It is helpful to recall the tension between modelling *status* and *events*, discussed extensively by Dix [49, chapter 10]. As Dix points out, there is an essential difficulty in modelling status, a state-based notion, in terms of an event based model. The required relationship between the display status and its future behaviour breaks down between an input and an output interaction. Dix questions the feasibility of abstractions that model both input and output by events, and suggests that an 'interactivity condition' should be applied, e.g. that an output follows all input. This condition can be enforced trivially as a temporal constraint on the ADC interactor, e.g. the continuous feedback constraint of section 6.1.1, but it does not solve the problem. Rather, the interactivity condition should be associated with the similarity predicate.

Sufrin and He [169] and by Abowd [1] model input as events, and the result and display as states. In both cases, the underlying representation of their model is state based. This makes the comparison straight forward, but there are restrictions on the type of interactions that can be modelled. In the examples of Sufrin and He [169], all user input is directly associated with an instant update of the display information.

To weaken the formal expressions of this section, the similarity predicate, of equation 7.2, should only apply to a subset of the states of the interactor. For example, it could apply to all states that follow an output, a weaker requirement, or to states that may proceed an input, a stronger requirement. The latter ensures that the relevant properties hold whenever the user is able to influence the behaviour of an interactor. Therefore the similarity predicate is defined now as follows:

$$\begin{aligned} &similar_A(P,Q) \text{ iff} \\ &\quad out_{aout}(P) = out_{aout}(Q) \neq \quad out_{dinp}(P) \neq \quad out_{dinp}(Q) \neq \\ &similar_D(P,Q) \text{ iff} \\ &\quad out_{dout}(P) = out_{dout}(Q) \neq \quad out_{dinp}(P) \neq \quad out_{dinp}(Q) \neq \end{aligned} \qquad 7.6$$

7.1.1 Example: Predictability of the scrolling list

Consider the scrolling list of section 4.8. Let P be a state of the list interactor, reached after the trace below, where the interactor is initialised with values LA and LD.

1 start
2 l_dinp ?p1:pnt;
3 l_dout !echo(p1, LD, LA);
4 l_dInp ?p2:pnt;
5 l_aOut!result(input(p2,input(p1,LD, LA),echo(p1,LD, LA)));

Different instantiations of this trace are obtained by giving values to the value identifiers p1 and p2. When a point is input at line 2, by the definitions of *ls_adu* and *ls_ad* of section 4.8, the abstraction of the interactor becomes:

$$\text{input(p1,LD, LA)} = \text{sel(LA, pick(LD, p1))} \tag{7.7}$$

An *l_dout* interaction follows, at line 3. This changes the display state to:

$$\text{echo(p1,LD, LA)} = \text{changeLne(LD,pick(LD,p1))} \tag{7.8}$$

At line 4, the second input event changes the abstraction to

$$\begin{aligned}&\text{input(p2,input(p1,LD, LA),echo(p1,LD, LA))} = \\ &\text{sel(sel(LA,pick(LD,p1)), pick(changeLne(LD,pick(LD,p1)), p2)} = \\ &\text{sel(LA, pick(LD,p2))}\end{aligned} \tag{7.9}$$

Consider p1 and p2 such that different elements of the list are selected:

$$\text{pick(LD,p1)} \neq \text{pick(LD,p2)} \tag{7.10}$$

Let the state Q be reached after the same trace as P, with the omission of the second input action at line 4. Then the display of Q is the same as that of P, but different results will be read from their *aout* gates.

$$\begin{aligned}&\text{out}_{\text{l\_dout}}(P) = \text{out}_{\text{l\_dout}}(Q) = \{\text{l\_dout!changeLne (LD,pick(LD,p1))}\} \\ &\text{out}_{\text{l\_aout}}(P) = \{\text{sel(sel(LA,pick(LD,p1)),pick(LD,p2))} \\ &\text{out}_{\text{l\_aout}}(Q) = \{\text{sel(LA,pick(LD,p1))}\end{aligned} \tag{7.11}$$

This interactor is not honest and also not strong WYSIWYG since

$$\begin{aligned}&\text{similar}_{D}(P,Q) \quad (\neg\, \text{similar}_{A}(P,Q)) \\ &(\text{out}_{\text{dinp}}(P) = \{\text{l\_dinp?p3:point}\} \neq \quad) \\ &(\text{out}_{\text{dinp}}(Q) = \{\text{l\_dinp?p2:point}\} \neq \quad)\end{aligned} \tag{7.12}$$

P and Q are not display equivalent. This can be seen if the traces leading to P and Q are suffixed by an output event on the display side. For P, the next output event on the display side is:

(6-P) l_dout !echo(p2, sel(LA,pick(LD,p1), echo(p1, LD, LA)

For Q an output event on the display side is:

(6-Q) l_dout !echo(p1, LD, LA)

From the specification of the data type *ls_ad*, in section 4.8, it follows that

echo(p2, sel(LA,pick(LD,p1), echo(p1, LD, LA)) =
changeLne(changeLne(LD,pick(LD,p1)),pick(changeLne(LD,pick(LD,p1)),p2)=
changeLne(LD, pick(LD, p2)) 7.13

which is not equal to the value output with the event (6-Q)

echo(p1, LD, LA) = changeLne(LD, pick(LD,p1)) 7.14

However if the controller component, of section 4.8 is modified so that an output is effected immediately after each input action, i.e. by enforcing the constraint called continuous feedback in chapter 5, then such differences can not occur. The improved controller component is as follows:

```
process CC [dinp, dout, ainp, aout] : noexit :=
    ainp?X:lstel; dout?X:scrLst;          CC[...] []
    aout?X:el;                            CC[...] []
    dout?X:scrLst;                        CC[...] []
    dinp?X:int;        dout?X:scrLst;     CC[...]
endproc
```

The traces leading up to states P and Q are identical for the revised interactor. The similarity condition can be true only after interaction (6-P) or (6-Q) respectively, when it is again possible to accept input. With the revised interactor, the similarity predicate can only be true when

pick(LD,p1)) = pick(LD,p2) 7.15

If this is the case, it turns out that the two states are also display-equivalent, i.e. $D_P \approx D_Q$, and the revised interactor is display-predictable.

This example shows the kind of reasoning required to prove the properties discussed. It draws attention to that these properties depend both on the data type specification for the interactor and also the dialogue specification.

7.1.2 Validity of predictability and observability formalisations

The reader has been cautioned already in chapter 3 about the limited soundness of these formal expressions. While it has been argued that they describe some user interface design heuristics in the framework of the interactor model, they cannot support claims that the system will be considered predictable, honest, etc., by its users. The ADC interactor does not model how users perceive what is displayed, nor when two displays are judged to be the same. The same applies for the comparison of the display and the abstraction side behaviours. Whether two results are similar or not, and whether two

interactor states are abstraction equivalent or not, may depend on the user's tasks, the mental model that the user constructs of the system, the situated behaviour of the user, etc. A further limitation of the formulations of this section has been flagged already. It concerns the formal definition of equivalence as a means of comparing interactive behaviours. Observational equivalence is sensitive to differences evident after prolonged sequences of interactions. Clearly a human user is not always able to distinguish such differences. Following this arguments it is clear, that asserting a formal expression of similarity or equivalence of interactors is, in essence, an assertion of a psychological theory. Such a proposal is outside the scope of this thesis. However, it interesting to note the bridging role of the model for incorporating psychological motivations into a formal analysis.

7.1.3 Verification of predictability and observability properties

In general, automatic verification is impractical, because the sets out(P) and out(Q), of equation 7.6, may be infinitely large and processes P and Q can be infinite. The discussion below exemplifies some of the problems involved in the automatic verification of result predictability. Similar problems, if not at a greater extent, are encountered with the verification of the remaining properties discussed in this section.

For example, to show that an interface represented by an ADC interactor is result predictable, table 7.1 requires the following implication to be shown:

$$\text{similar}_A(P,Q) \quad A_P \approx A_Q \tag{7.16}$$

for any two states P and Q of the interactor. The practical difficulty of comparing the LTS of two different LOTOS behaviour expressions, i.e. of the interactor and its abstraction side behaviour, can be overcome by the observation that

$$\text{similar}_A(P,Q) \Leftrightarrow \text{similar}_A(A_P,A_Q) \tag{7.17}$$

Then it is straight forward to verify result predictability by a search on the LTS (S, Act, ¾→ , s_0) that models the behaviour expression A so that:

$$P,Q \quad S \mid \text{similar}_A(P,Q) \quad \text{out}(P) \neq \quad \text{out}(Q) \neq \quad (P \neq Q) \tag{7.18}$$

If no such states are found, then the interactor is result predictable.

The success of this venture depends on whether the model checking tools can generate a finite labelled transition system that models the specified behaviour. There are both theoretical and technological limitations. From a theoretical point of view a finite set of values should be produced by operation *result.* Even if this is the case, the comparison of the values of two ACT-ONE terms is a decidable problem in some cases only. Also, the size of the LTS produced may prohibit any analysis. Clearly, it is very important to choose sensibly the level of detail injected in the description. Tool support, e.g. Lite [30, 119], Caesar/ Aldebaran [68] may provide finite interpretations for LOTOS processes.

Tools apply mostly to basic LOTOS specifications, in which case they can verify the equivalence of two processes but they ignore the data component of interactions. When the data types are specified so that any inquiry operation returns a finite set of values, Caesar/Aldebaran can verify Full LOTOS behaviours as well. To achieve it in practice may still be a challenging task, because of the size of the LTS representations produced.

To establish the property of result predictability, equation 7.18 can be verified by a search on a single LTS. In the case of observability properties, like WYSIWYG and trustworthiness of table 7.2, the verification will need to compare two different LTS, one concerning the abstraction side behaviour and the other concerning the display side behaviour. This is difficult to do with the available tool support since there is no way to relate individual nodes of each derived LTS to the nodes of the LTS of the interactor. This is not a serious theoretical limitation, but it means that the general purpose verification tools for LOTOS do not readily support the verification of observability properties.

In summary, expressions like those of tables 7.1- 7.3 have an intuitive appeal and are concise formulations of heuristics for user interface design. Unfortunately, their practical utility is limited because they are hard to verify. It is probably for this reason that the publications that discuss this kind properties, e.g. [1, 169], concern small scale examples and use informal reasoning rather than rigorous or formal proofs. There has been no report of the automatic verification of this type of properties.

7.2 Dialogue analysis

This section concerns the formal verification of the dialogue component of an interface design. This verification presupposes the distinct formal specification of the dialogue. This is a trait of UIMS with a separable dialogue component, for which the special purpose dialogue specification notations, discussed in section 3.1, have been developed. The review of interface architectures, in chapter 2, showed that modern object based interface architectures are characterised by the distributed implementation of the dialogue. This means that the dialogue is encoded partly in each object and partly in the way the objects are composed together. This is true for interactor based specifications too, so the question that arises, is when and how can interactor specifications be used as dialogue specifications. Section 6.6 discussed this problem, and provided a characterisation of what is the scope of ‘dialogue’ in the context of the ADC model. Also, section 6.6 discussed how the dialogue specification for a group of ADC interactors is specified in the CU of their synthesis. The CU does not specify any data manipulations, so the transformation P(CU) to basic LOTOS models the dialogue sufficiently. A similar proposition was put forward for the abstract views of interactors, for which the dialogue specification is also a basic LOTOS process. Thus, even when there is not a separable dialogue component in the implementation software, the ADC model makes it possible to factor out the dialogue specification for the whole of the interface specification.

Some dialogue properties that a designer might wish to verify are the following:

1. Deadlock freedom. The interface does not reach termination because of an inconsistent dialogue specification. Checking for deadlock freedom is supported by model checking tools, e.g. [68]. When a deadlock is detected the tool provides a diagnostic, indicating the sequence of actions that lead to the deadlock.

2. Completeness. The specification has specified all intended and plausible interactions of the user with the interface. Completeness can be inferred only by comparing an interface specification to, say, some task specification, some requirements document, etc. For example, the designer might wish to specify that all possible action sequences necessary to achieve a certain task are supported by the interface design.

3. Determinism. A user action, in a given context, has only one possible outcome. Checking for determinism is supported by model checking tools, e.g. [68].

4. Reachability. A range of properties fall under this heading, including deadlock freedom and completeness. It qualifies the possibility and ease of reaching a target state, or set of states, from an initial state, or set of states.

Dialogue properties can be specified directly in LOTOS and they can be verified against the dialogue specification P(CU). This process is supported by the Caesar/Aldebaran toolset [68]. In many cases the formal expression of dialogue properties is facilitated by referring to some notion of state. This is consistent with the LTS interpretation of LOTOS, but the interactor specification does not support an explicit description of these states. It is possible though to specify states of the LTS by the actions they offer. A temporal logic can be used to specify dialogue properties, as propositions pertaining to the underlying LTS. This process is supported by the LITE [30, 46] and the AMC [70] tools. The discussion below presents both the specification of dialogue properties in LOTOS, and also the use of a temporal logic called ACTL [48]. ACTL is an action based and branching time temporal logic. Its operators use actions to specify the states of an LTS. ACTL provides quantification operators both for paths and for linear time. Each linear operator must be preceded by a branching one and vice versa. A brief introduction to ACTL is provided in appendix A.3.

7.2.1 Informal description of dialogue properties

First, dialogue properties are discussed informally in terms of a state based representation. A reachability property specification consists in three parts. The first specifies the state, or states, from which the transitions of interest may start. The second part qualifies the path, i.e. it may specify some invariants that must hold for all intermediate transitions, or some temporal operator that defines when the transition is possible, e.g. never, always, sometimes, etc. The third part describes the target state or set of states. This scheme, has been used in [30, 142, 143] and it provides a concise a classification of dialogue properties. The presentation below is very similar to that of

Shorthand Label	Path Qualifier	Start State or Set of States	Target State or Set of States
State Accessibility	Possibly	Initial state	A specified state or set of states
Deadlock Freedom	Never	Initial state	A state offering no actions
State Exclusion	Never	Initial state	Any state that does not satisfy an unwanted condition
Action Accessibility	Always: A set of actions should be available	Initial state	Any state
Action Succession and Integrity Constraints	Always: a predicate on the sequence of actions should hold	Initial state	Any state
State Floatability	Eventually, without entering an undesired state	A specified state q	A specified state r
Weak State Inevitability	Eventually	A specified state q	A specified state r
Strong State Inevitability	Eventually	Any state	A specified state r
Reversibility	Eventually	At any state that follows action α	A state identical to the one before α fired
Re-startability	Eventually	Any state	Initial state

Table 7.4. A state-based scheme for the classification of dialogue properties

[143], with the difference that the dialogue specification is not assumed to be a refinement of some user task model. This does not change the range of the formal expressions but it means that they are interpreted differently. The dialogue properties discussed here pertain only the designed interface as opposed to characterising its relation to a task. In section 7.4, the relationship with the task model is also discussed. Finally, the description of table 7.4 puts more emphasis than [143] on the temporal operator associated with the path qualifier. This is consistent with the formalisation in ACTL that follows.

The properties summarised in table 7.4 can be instantiated for the case of the Simple Player™ specification discussed in chapter 5. An example of *action accessibility* for this interface is that it should always be possible to resize the movie window or to change the volume. The start state for this property is the initial state of the LTS, there are no conditions on the target state, and the path qualifier is that the action resize should be available at all times. So from any state, it should be possible to issue a resize command, i.e. to invoke a *dinp* command on the resize box interactor. This property does not hold, simply because the resize interactor requires a *dinp* interaction to be followed by an output on a gate *dout*, i.e. by some feedback. An example of an *integrity constraint*

could be that the whole of the interface cannot issue a stop command twice, without an intervening play command, or that the interface does not accept successive double clicks as input.

The descriptions of table 7.4 may help for an informal specification of dialogue, and possibly an informal testing of some of these properties. These descriptions can apply to the whole behaviour of the ADC interactor or only the dialogue specification P(CU). Clearly, it is important to distinguish the two. For example, by definition, all interactor dialogues in the case study are restartable in that their dialogue specification P(CU) is restartable. However, this is true only because the dialogue specification does not refer to the data component of the specification. The specification of the case study did not model re-initialisation of the movie interface, which is what this property might be taken torefer to. The scope of the dialogue properties discussed below concerns only the CU specification, and the verification methods follow this assumption.

Olsen et al. [143] proposed the specification of the properties of table 7.4, in terms of propositional production systems (PPS). They have also outlined algorithms for the assessment of the properties they discuss. Abowd et al. [3] suggested that the PPS specification of an interface dialogue should be transformed to a state transition network. Dialogue properties would then be specified in a branching time temporal logic and would be evaluated using a model checking tool. Neither approach has been implemented. At this point the advantages of using a more established formal specification language, like LOTOS, become apparent. In the following paragraph, two recent approaches to specifying and verifying dialogue properties are summarised and compared. Both use established formal methods and general purpose tool support, as opposed to tools specifically created to support the interactor model.

7.2.2 Formal specification of dialogue properties

Palanque and Bastide [144] describe the specification of user interfaces using an object based variant of Petri-Nets. Their notation is interpreted to a standard Petri-Net specification, and dialogue properties such as absence of deadlock, re-initialisability, computation of reachable states, etc., are specified in terms of this underlying representation. Verification can apply to the whole system or to some of its components [147]. For example, Palanque and Bastide [144] specify the predictability of a command as a reachability property. A command is called predictable if its target state is always the same regardless of what its starting state is. A limitation of their approach is that the specification of user interface properties is not assisted by an architectural model or a higher level language for describing properties of interaction.

Paternó [151] uses ACTL to specify dialogue properties of interfaces specified in LOTOS, in the framework of the Pisa interactor model [150]. Mezzanotte and Paternó have applied this approach to the verification of a multi-modal flight information system [152] and a graphical interface to an air traffic control tool [153]. The LOTOS specification of an interface is transformed to basic LOTOS using the LITE specification

environment [30]. When possible, theoretically and practically, an LTS representation of the interface is produced. A model checking tool [70] can then verify whether the LTS satisfies the ACTL expressions. The important difference to the work of Palanque and Bastide discussed earlier, is that the specification of dialogue properties need not refer to the underlying LTS representation. The interactor structure prompts the expression of the relevant properties. Some examples of dialogue properties specified in ACTL are discussed below, 'translated' in the framework of the ADC interactor model. The purpose of these examples is to illustrate what looks to be the most advanced approach to the verification of dialogue properties and to show that it applies also to the ADC model. Some caveats are flagged and the discussion draws attention to benefits of the ADC model as a representation of the interface design.

- An input action on the display side may generate an effect on a specific part of the interface.

  $$AG([dinp]\ E(tt_{tt}U_{aout}tt)) \qquad 7.19$$

This can be interpreted as follows. Always (the AG operator), from the state following a *dinp* interaction, there exists (the E operator) a sequence of interactions, for which the only requirement is that they lead up to an interaction on a gate *aout.* This sequence of interactions is specified by the path formula $tt_{tt}U_{aout}tt$.

In expression 7.19, the user interaction and the output interaction are both described by what was called a *role* of a gate in section 6.1. The role *dinp* characterises a gate used for input on the display side. Expression 7.19 is in fact a template, that can be instantiated by substituting this description with an appropriate pair of gates of the interactor studied. For example, they could be gates *l_dinp* and *l_aout* of the scrolling list of section 4.8. In this way, the interactor structure prompts generic expressions of dialogue properties. This is shown by a further example.

- A user input on the presentation side may eventually result in a feedback event.

  $$AG([dinp]\ E(tt_{tt}U_{dout}\ tt) \qquad 7.20$$

For example, expression 7.20 can be instantiated for the list interactor of section 4.8 by substituting the relevant gate identifiers:

$$AG([l\_dinp]\ E(tt_{tt}U_{l\_dout}\ tt)) \qquad 7.21$$

This means that an input on the display side of the list interactor may result, eventually, in a selection of a member of a list being output on the application side.

Model checking applies to a transformation of the specification to basic LOTOS and not to full LOTOS. Therefore the formulae above are weak specifications of the properties they model. Expression 7.21 for example, does not ensure that the value sent to the application at gate *l_aout* is the one that results from the input operation.

- Continuous feedback.

$$AG[dinp]\ A(tt\ {}_{\neg dinp}U_{dout}tt) \qquad 7.22$$

This can be interpreted as follows. Always (the AG operator), if a sequence of interactions ends with an interaction on a gate *dinp* (the term [dinp]) then for all possible evolutions (A operator) the following condition will hold: no interaction on a gate *dinp* may take place before an interaction on a gate *dout* takes place.

Alternative characterisations of gates, that are not specific to the interactor model, may help write dialogue specifications. For example, assuming a class of events *err* that signify an error, then error recovery may be specified:

- Error events are eventually followed by input to the application

$$AG([err]\ E(tt_{tt}U_{aout}tt) \qquad 7.23$$

The truth of the above expression is only an indication that the system will recover after such an error, since it offers output on its application side.

- Existence of messages explaining user errors

Assuming a set of gates associated with explanation messages to the user, e.g. a subset of the gates G_{dout}, then it is possible to describe the response of a system to an error:

$$AG([err]\ E(tt_{tt}U_{explain}tt)) \qquad 7.24$$

In a similar fashion, if some events are associated with task completion, then the formal specification may be used to verify whether it is feasible for the user to complete their tasks for a given interface design. The link to task modelling, the identification of errors, the identification of errorful behaviour, etc., are not part of the interface specification. The validity or not of the predictions arrived at depends on how these gate sets are defined. This definition may follow theoretical considerations or design decisions that are not part of the interactor model.

What is most interesting about the approach of Paternó and Mezzanotte is how they capitalise on the standard structure of the Pisa interactor model to develop reusable expressions of interaction properties. The expressions they propose, which are similar to the examples listed, refer to sets of gates of an interactor: gates for user input, for display, for communicating to the application, etc. These gate sets are easily identifiable for a single interactor. A specifier can easily instantiate the general property definitions, substituting gate identifiers or sets of them in the relevant expressions. For an interface specified as a composition of interactors, in the framework of the Pisa model, these gate sets are defined by inspecting the configuration of the interactors [152]. In the framework of the ADC interactor, a complex configuration can be transformed, by synthesis, to a single interactor. During synthesis the role of the gates is redefined as was described in section 6.1. As a result, the formal specification of dialogue properties is the same for a simple interactor as it is for a compound interactor.

The work of Paternó and Mezzanotte in the analytical evaluation of user interface specification is interesting also because of the scarcity of results in the automated verification of user interface specifications. They demonstrate the potential of automating much of the specification and verification tasks and using formal methods in the design of user interfaces. This is an important contribution towards the wider application of formal methods techniques in user interface design, particularly by non-formal methods experts. Their reports though can be criticised for overplaying this potential. While the Pisa model is specified in full LOTOS, the verification concerns only basic LOTOS. The Pisa model, does not have an 'orthogonal' description of data handling and temporal ordering properties. Thus, in the framework of the Pisa interactor model, dialogue specifications may well depend on the value of the state parameters of the Pisa interactor. Simply 'stripping' the data component by the transformation to basic LOTOS does not preserve the dialogue specification. Paternó [151] and Paternó and Mezzanotte [152, 153] overlook these limitations, making too strong a case for the use of model checking tools.

In conclusion, there is much merit to the use of a temporal logic to specify dialogue properties of interaction. Generic expressions of dialogue properties follow by inspection of the interactor specifications. Powerful tool support helps their automatic verification. Requirements external to the interface architecture, e.g. task related behaviour, error handling, etc., can also be specified in the same fashion. The ADC model can be used in the manner introduced with the Pisa model. Also, it supports the articulation of dialogue properties for synthesised interactors and provides a clearer definition of the dialogue scope.

7.2.3 Constructive specification of properties with LOTOS

As mentioned already, dialogue properties can be specified directly as LOTOS specifications. An immediate practical advantage is that the same language is used to specify ADC interactors and their properties. The dialogue specifications are very similar to those used inside the CC of the interactor, so the same set of processes can be used for both purposes, e.g. they could be taken from some 'library' of pre-defined dialogue properties. The verification is supported by the CAESAR/ALDEBARAN toolset [68]. Perhaps not surprisingly, some properties can be easier to specify in LOTOS rather than in ACTL and vice versa.

The LOTOS process below specifies that an occurrence of an input on the display side is followed by the occurrence of an output before another input is allowed. An output event is still allowed to happen without a prior input event.

```
process feedback[inp, out] : noexit :=
    inp; out; feedback[inp, out]
[]  out; feedback[inp, out]
endproc
```

This process definition is similar to the constraints specified in the CC component. To specify that an input event is possibly followed by an output event, the interactor list is

compared to the process feedback. For this comparison all actions that are not input or output on the display side are hidden, since they do not concern this property. This is specified as an argument for the verification as follows:

```
hide all but
l_dinp
l_dout
```

The list interactor is compared to the feedback property with respect to weak observational equivalence:

hide (G - {l_dinp, l_dout}) in P(list_CU) ≈ feedback[l_dinp, l_dout] 7.25

Weak observational equivalence is chosen because it is recognised that each interactor may exhibit internal behaviour which is ignored and because all actions apart from *l_dinp* and *l_dout* are hidden. The hiding means that the property holds even if the feedback is not offered immediately after the input interaction. Equivalence 7.25 describes the same behaviour as 7.20. It is difficult to assert, unless to describe a subjective preference, which approach is easier to construct or to comprehend.

If the feedback property is amended so that other interactions are not hidden, then continuous feedback may be specified as follows:

```
process continuous[inp, out, other] : noexit :=
    inp; out; continuous[inp, out, other]
[]  out;      continuous[inp, out, other]
[]  other;    (continuous[inp, out, other][]stop)
endproc
```

The list interactor is compared to the process continuous, up to a renaming of all gates other than *l_dinp* and *l_dout* to the gate label other. The renaming relation is specified as follows:

```
rename
"l_aInp"     ->     "other"
"l_aOut"     ->     "other"
"start"      ->     "other"
"suspend"    ->     "other"
"resume"     ->     "other"
"abort"      ->     "other"
```

Then the following relation needs can be verified using the Aldebaran tool, up to the renaming above:

P(l_cu) ≈ continuous[l_dinp, l_dout, other] 7.26

The model checking will return a verdict false in this case, and true if the constraint component is modified as in paragraph 7.1.1. Again the verification of 7.26 can be compared to 7.22. The renaming and the hiding operations define an abstraction relation. Interactions can be characterised as input or output by the abstraction relation. Depending on how these arguments are defined the same process specification for

continuous feedback may apply at different sets of interactors and at different levels of abstraction.

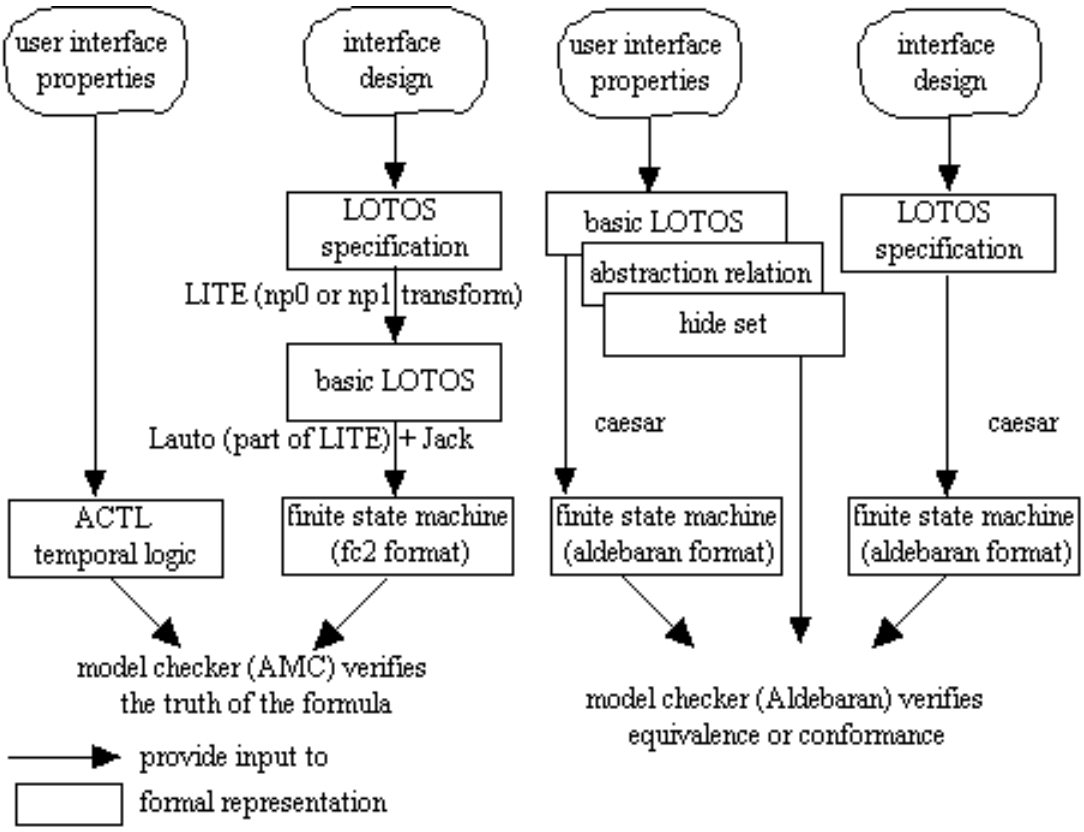


Figure 7.1. Illustration of two possible approaches to model checking of LOTOS specifications. In the left, using the AMC tool and in the right using the Caesar/Aldebaran toolset.

LOTOS is very well equipped to define action succession properties and safety properties. ACTL is certainly more concise. The higher level ACTL constructs, e.g. AG (always), [dinp]φ, etc., are quite powerful constructs that some people may find more intuitive than a process algebraic description as, for example, is argued in [46]. Both approaches discussed require a good understanding of LOTOS and ACTL respectively and a careful interpretation of the meaning of the specifications written. Figure 7.1 outlines the two different approaches for model checking using the AMC tool [70], or the Caesar/Aldebaran toolset [68].

7.2.4 Conclusion

This section has discussed interactor models as a conceptual framework for specifying and verifying dialogue properties. Rather than requiring an in depth understanding of a monolithic representation of a dialogue, interactor models prompt the expression of dialogue properties in terms of architectural constructs, i.e. the gates of the interactors and sets of such gates with a common purpose, e.g. input, output, controlling the functional core, etc. The verification of these properties though, needs to apply to a distinct dialogue specification with a clearly defined scope. This dialogue specification, does not need to correspond to a separable dialogue implementation. Chapter 6 provided

a clear description of dialogue in ADC interactor specifications and an approach for factoring out its specification from a composition of many nteractors.

The approaches discussed rely heavily on the use of general-purpose verification tools. It is hoped, that as this technology evolves and matures, it will benefit more the user interface designers who construct formal specifications of their designs. This section discussed the use of two general purpose model checking tools for behavioural specifications and a scheme for describing such properties informally. The specifications were independent of assumptions regarding the user behaviour or cognition and it did not aim to propose a set of principles for dialogue design. Emphasis was drawn to the careful interpretation of dialogue specifications.

The approach to dialogue analysis discussed above, is quite traditional, in that it concerns the temporal ordering of interactions. It is worth noting, that the LTS interpretation of the interface specification is also amenable to alternative types of analysis. Thimbleby [171] discusses how graph analysis may be used to provide quantified predictions relating to the usability of a system, that are also neutral with respect to implementation platform or user considerations. For example, these may provide estimates of the complexity of a task, how much the user needs to learn to operate the system, or as reported in [7], whether or not a system is well structured. The successful use of these alternative formulations of usability, as for all the properties discussed in this chapter, depends on choosing carefully the level of description of the system.

7.3 Top-down interface design and the ADC interactor model

This thesis has avoided making assumptions regarding user cognition and behaviour, the phenomenon of interaction, and it has not endorsed any particular view of the design process, e.g. top-down vs. bottom-up, task based, using style guides, etc. If only to show how the interactor model may be used in conjunction with such theories and methods, it is necessary to extend the discussion to these topics. This section discusses how the ADC model can support the general concept of stepwise refinement. In the following section, a more psychologically informed method is discussed that involves modelling some aspects of user knowledge. The purpose of this exercise is to draw links to the relevant bodies of research and to illustrate the versatility of the ADC model, rather than to advocate a particular view of design.

Stepwise refinement is a process by which software is developed incrementally through a series of distinct design steps. In each step design options are resolved adding detail and structure to the design representation. Top-down design methods prescribe a process which tries to ensure that the original and the resulting design specifications are consistent. In a formal method of software development this consistency requirement is a formally defined relationship, e.g. some form of equivalence, or a formally defined implementation relationship. Below, the terms *specification* and *abstract implementation* (or simply implementation) are used in reference of two successive

design representations, where the latter is a refinement/implementation of the former. This terminology is standard for discussing LOTOS specifications [158]. The user interface software that results from the refinement process is called the *realisation* of the interface specification.

An interface specification may be transformed to its implementation using any of the following transformations:

- Functional Decomposition. A single interactor is decomposed to a group of interactors with the same externally observable behaviour.
- Functional Rearrangement. A group of interactors is transformed to a different group of interactors with the same externally observable behaviour.
- Functional Extension. A given specification is enriched with extra functionality that is consistent with the original specification. For example, a help facility may be added on an interface as an extension of the original functionality.
- Functional Reduction. A specification may include design options that are left unresolved. An implementation of the specification may be obtained by fixing one or more of these options.

To support these transformations in the context of the ADC interactor model entails that the outcome of each should still be an ADC specification or, in the case of decomposition and rearrangement, a behaviour expression that combines ADC specifications. It is trivial to see that an interface specification can be reduced or extended within the framework of the ADC interactor model. For example, increasing the gate set and specifying relevant behaviours, is an extension of the functionality, which is consistent with the original specification. On the other hand, adding a constraint on the CC of an interactor is a functional reduction of it. Functionality decomposition is supported by the decomposition transformation of the previous chapter, while rearrangement may be effected through a combination of synthesis and decomposition. The example that follows illustrates the refinement of an interactor specification by the decomposition transformation of chapter 6. The example discusses a hypothetical scenario concerning the specification of Simple Player™. As in chapter 5, the specification is reversely engineered to match the observed behaviour of the software. The specification is assumed to proceed through a series of abstraction levels which add structure and detail incrementally. This is an ‘imaginary design trajectory’, presented in [126] in a more extensive and informal manner, but it is not suggested that this was the actual process followed in the case study. A single step of this design process is isolated below, to discuss the application of the decomposition transformation of chapter 6.

The point of departure for this design step is a specification of the whole interface as a single interactor *spec*, which interacts with the user on its display side and with the application on its abstraction side. The specification for this design step incorporates several design decisions already. The functional core has been specified as a boot

strapping activity for the specification, as mentioned in chapter 5, so the gates of the interface which are connected with the functional core are known in advance. The specification describes the temporal constraints on the input, reflecting the properties of the interaction toolkit used. The manipulation of the volume and movie box parameters is specified as a well formed ADU, called *specADU*. The temporal constraints for their management are encoded in the constraints component *specCC* of the interactor. Interactor *spec* is summarised below and it is illustrated in figure 7.2. For brevity all mouse input is grouped under the heading *mouseInput* and all keyboard input under the heading *keyboardModifiers*. On the display side one output gate is allocated to each interactor with a display component. In figure 7.2, the display output gates are grouped under the heading *display*. Arrows indicate gates where data is communicated with the interaction and simple lines indicate simple synchronisation.

The interactor *spec* of figure 7.2 is defined as follows:

```
process spec[...]: noexit :=
specADU[pressVOL, moveVOL, releaseVOL, doutVol, getVolume, setVolume, pressBox, moveBox,
                                                                    relBox, doutBox, setMovieBox]
    |[pressVOL, moveVOL, releaseVOL, doutVol, getVolume, setVolume, pressBox, moveBox, relBox,
    doutBox, setMovieBox]|
specCU[...]
endproc
```

Process *specADU* is a well formed ADU that is the parallel composition of two elementary ADUs, respectively managing data of type *volume_ad* and *resizeButton_ad*. The definitions of *volADU* and *rbADU* and the corresponding data types is included in appendix A.4 for completeness.

```
process specADU[pressVOL, moveVOL, releaseVOL, doutVol, getVolume, setVolume, pressBox,
                        moveBox, relBox, doutBox, setMovieBox]:noexit:=
volADU[pressVOL, moveVOL, releaseVOL, doutVol, setMovieBox, getVolume,setVolume]
            (preferredVolume,defaultVolumeBar, defaultVolumeBar)
    |[setMovieBox]|
rbADU[pressBox, moveBox, relBox, doutBox, setMovieBox](graphicalContext, resizeBox, resizeBox)
endproc
```

The controller unit *specCU* is constructed according to the general definition of the CU described in section 6.7. The definition of the constraints component *specCC* is presented below.

```
process specCC[...] : noexit :=
noConstraint[video, stills, play, pause, getTime, gotoTime,setSelection, gotoStart, gotoEnd, out_ok]
    |||
((volSeq[pressVOL, moveVOL, releaseVOL, doutVol, setMovieBox, getVolume, setVolume]
    |[ setMovieBox]|
rszSeq[pressBox, moveBox, relBox, doutBox, setMovieBox])
    |[pressVOL, moveVOL, releaseVOL, pressBox, moveBox, relBox]|
inputConstraints[...])
endproc
```

First, the temporal constraints for user input are discussed. The mouse input may be clicking, pushing and holding, or dragging. Each of the components which are visible on

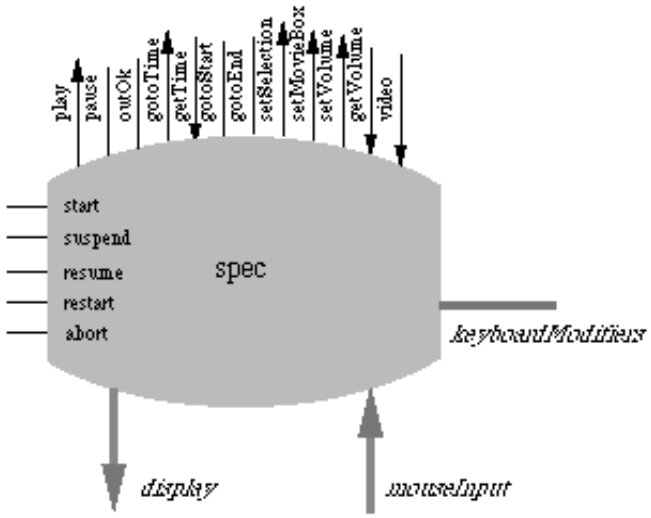


Figure 7.2. Simple Player™ modelled as a single ADC interactor. Grey lines and arrows represent groups of gates, described by the relevant heading in italic print.

the screen is manipulated in either of these ways and for each some mouse input constraint applies. For example, any number of moves of the mouse may take place when this is dragged. While a push button is held pressed, it can be moved out and back into an 'active region' surrounding the button, as described by the toggle process. The following process describes these temporal constraints on the mouse input:

```
process inputConstraints[click, doubleClick, press_shift, move_shift, release_shift, pressPLR,
      moveTHMB, relPLR, pressVOL, moveVOL, releaseVOL, pressFwd, relFwd, moveInFwd,
      moveOutFwd, pressBT, moveInBT, moveOutBT, releaseBT, pressBwd, relBwd, moveInBwd,
      moveOutBwd, pressBox, moveBox, relBox, pressRate, moveRate, relRate]: noexit:=
(   mouseButton[click, doubleClick]
|||  drag[press_shift, move_shift, release_shift]
|||  drag[pressPLR, moveTHMB, relPLR]
|||  dragV[pressVOL, moveVOL, releaseVOL]
|||  hold[pressBT, moveInBT, moveOutBT, releaseBT]
|||  hold[pressFwd, moveInFwd, moveOutFwd, relFwd]
|||  hold[pressBwd, moveInBwd, moveOutBwd, relBwd]
|||  drag[pressBox, moveBox, relBox]
|||  drag[pressRate, moveRate, relRate])
where
    process mouseButton[c,d] : noexit :=
            c; mouseButton[c,d] [] d; mouseButton[c, d]
    endproc
    process dragV[p, m, r] : exit :=
            p; (repeat[m] [> r?x:pnt; dragV[p, m, r])
    endproc
    process drag[p, m, r] : exit :=
            p?x:pnt; (repeat[m] [> r?x:pnt; drag[p, m, r] )
    endproc
    process hold[p, mi, mo, r] : exit :=
            p; (toggle[mo, mi] [> r; hold[p, mi, mo, r])
    endproc
    process toggle[m, n]: noexit :=
```

```
            m; toggle[n,m]
    endproc
    process repeat[m] : noexit :=
            m?x:pnt; repeat[m]
    endproc
endproc
```

This specification of the input constraints demonstrates how the corresponding process definitions are re-used, a point discussed already in section 5.8. To achieve the synchronisation with the ADU, their gates are typed, necessitating the repetition of some process definitions. For example, the process *drag* was defined in two forms, one where the action *press* is a pure synchronisation action and then again where it is associated with a parameter of sort *pnt*. In this case LOTOS hinders the specification activity by being too strict and some pre-processor to support the polymorphic definition of these constraints would be helpful.

Resizing may occur at any moment during interaction. While the user drags the resize box, the window for the application is not resized continuously. Instead an outline box is given as an intermediate feedback. When the operation is completed with a release of the mouse button, all interactors receive the updated coordinates through the gate *setMovieBox,* and adjust their displays. This is specified as follows:

```
process rszSeq[pressBox, moveBox, releaseBox, doutBox, setMovieBox]: noexit :=
    pressBox?x:pnt; doutBox?y:pb_dsp; rszSeq[...] []
    moveBox?x:pnt; doutBox?y:pb_dsp; rszSeq[...]  []
    releaseBox?x:pnt; doutBox?y:pb_dsp; setMovieBox?x:rct; rszSeq[...] []
    doutBox?y:pb_dsp; rszSeq[...]
endproc
```

The volume interactor is a slider that pops up with a mouse press. Its output informs the user about the current volume setting and allows the user to modify it. It pops down when the user releases the mouse button. When the mouse is first pressed, the current setting of the volume in the functional core must be read first. This may be either a value set with a previous interaction or the preferred volume for the movie played.

```
process volSeq[pressVol, moveVol, releaseVol, doutVol, setMovieBox, getVol, setVol]: noexit :=
    pressVol;getVol?x:Int; doutVol?x:volumeBar;volSeq[...]
[]  moveVol?x:pnt; doutVol?x:volumeBar; setVol?x:Int;   volSeq[...]
[]  releaseVol; doutVol?x:volumeBar;                    volSeq[...]
[]  setMovieBox?x:rct;                                  volSeq[...]
[]  doutVol?x:volumeBar;                                volSeq[...]
endproc
```

The gates of the interactor which are not related by the constraints discussed so far, e.g. *setTime, gotoTime,* etc., are allocated to process *noConstraint*. This process specifies no constraint on the occurrence of interactions on its gates and is combined with the other constraints by the interleaving operator.

By the corollary of section 6.5, interactor *spec* can be decomposed into the synchronous composition of interactors *volume, resizeBox* and *noConstraintsCU* (as illustrated in

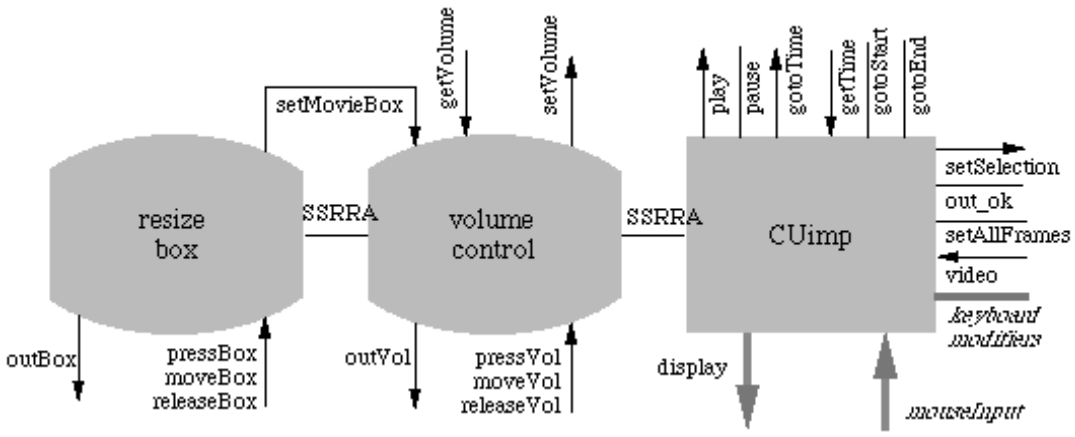


Figure 7.3. Decomposition of *spec* into a synchronous composition expression.

figure 7.3). The distributed form for this decomposition, i.e. the implementation for this refinement step, is specified by the following behaviour expression:

```
volume[start, suspend, resume, restart, abort, pressVOL, moveVOL, releaseVOL, doutVol,
                                                    setMovieBox, getVolume, setVolume]
    |[setMovieBox, start, suspend, resume, restart, abort]|
rszBox[start, suspend, resume, restart, abort, pressBox, moveBox, relBox, doutBox, setMovieBox])
    |[start, suspend, resume, restart, abort]|
CUimp[...]
```

The ADU of the two interactors are the processes *volADU* and *rbADU* defined previously. The CUs have the standard structure of section 6.5, and their constraints components are defined as follows:

```
process volCC[press, move, release, dout, setMovieBox, getVol, setVol]:noexit:=
volSeq[press, move, release, dout, setMovieBox, getVol, setVol]
    |[press, move, release]|
dragV[press, move, release]
endproc

process rszBoxCC[pressBox, moveBox, releaseBox, doutBox, setMovieBox]: noexit :=
rszSeq[pressBox, moveBox, releaseBox, doutBox, setMovieBox]
    |[pressBox, moveBox, releaseBox]|
drag[pressBox, moveBox, releaseBox]
endproc
```

Interactor *CUimp* is a standard CU whose constraints component *impCC* contains those behaviours from the process input constraints which have not been absorbed in the volume and resize box interactors.

```
process impCC[...] : noexit :=
noConstraint[video, stills, play, pause, getTime, gotoTime,setSelection, gotoStart, gotoEnd, out_ok]
|||
(   mouseButton[click, doubleClick]
|||  drag[press_shift, move_shift, release_shift]
|||  drag[pressPLR, moveTHMB, relPLR]
```

```
|||  hold[pressBT, moveInBT, moveOutBT, releaseBT]
|||  hold[pressFwd, moveInFwd, moveOutFwd, relFwd]
|||  hold[pressBwd, moveInBwd, moveOutBwd, relBwd]
|||  drag[pressRate, moveRate, relRate])
where (* ...as before...*)
```

Two interesting issues are highlighted by this brief example. First, that the decomposition transformation is not as widely applicable as would be desirable. This issue has already been discussed in section 6.4. In the context of this example, it is easy to see that a more plausible description for the original ADU would not be as a parallel composition of two components, but as a single process that manages both the parameters pertaining to the volume and those pertaining to the movie box. In such a case, the decomposition transformation would not apply because the volume data type uses the result of the resize box, and the condition 6.45 for the decomposition of an elementary ADU with multiple state parameters would not hold.

The second issue that arises is that the specifications are quite sizeable and the 'logistics' of carrying out the transformation step are not negligible. Tool support can relieve interface designers of such tedious tasks enabling them to focus on the creative part of their work. Current tools for developing LOTOS specifications support a set of transformations applicable to limited subsets of LOTOS and only for particular forms of the initial specification [20, 118]. Using this set of transformations is feasible, as indeed has been illustrated by Bernadeschi et al. [16]. The benefit of their approach is that they use standard and developed technologies. The drawback, which is relevant to this thesis, is that the concept of the interactor does not enter into the design process until the last stage, when the architecture of the system has been settled. What is advocated with the development of the ADC interactor, is that the concept applies to any abstraction level and at any stage in the design. For this to be a workable proposition, the development of special purpose transformations and tools to support them is necessary.

Bernadeschi et al. [16] start from a basic LOTOS specification of a 'black box' view of the whole interface, incorporating some temporal constraints, that pertain to the user tasks the system will support. This basic LOTOS specification of a task model, which is refined by successive transformations into an architectural description. Only when the architecture has been determined, the lowest level processes are refined to interactor specifications, using the Pisa interactor model of [150]. This design method assumes that the interface model should be a refinement of a task model and considers interactors only as the elementary building block for putting together the interface specification. This question regarding the relation of an architecture to the task model is discussed extensively in the next section.

The most important contribution of Bernadeschi et al. is to demonstrate how existing tool support and theory can serve the design of an interactive system. Their approach is still at an early stage of its development, i.e. there have been no reports as to the use of this approach within a realistic project, but it shows the potential to be scaled up. A similar approach to that of [16], using the ADC interactor model, is outlined in [126]. There, the emphasis is on the preservation of dialogue specifications, through a series of

decompositions of interactors and abstract views. Compared to the example shown here, that early discussion did not adequately demonstrate that the ADC model can be used as a design representation at all steps of the refinement process. Rather, like [16], the refinement concerns the basic LOTOS specification but, unlike [16], it did not use standard transformations and tool support. This observation raises some questions regarding the direction of future research. Throughout this thesis the use of standard notations and tools has been advocated. However, to reap practical benefits from the compositionality of the model, it seems that special purpose tools and possibly higher level notations, have to be developed. If such an approach is taken, it could well be at the cost of losing the benefits of future developments in the LOTOS language and tool support.

7.4 Relating interactor specifications with a task model

In the domain of human-computer interaction (HCI), the term *task* refers to the intentional activity of a user who interacts with a computer in order to achieve some goals. Tasks are studied by a *task analysis* process. The product of this process is a summative description of the tasks which is called a *task model*. Research in HCI has developed several task analysis methods and task model representations which vary according to their intended use and scope of application. An overview of this type of research can be found in [108]. This section discusses a formal representation for a task model that focuses on the temporal ordering of the task related activity of users. The model is based on Johnson's theory of Task Knowledge Structures (TKS) [108, 109 and 110]. Relating the task and the interactor formal models helps formalise some intuitions underlying task based design approaches. *Task based design* prescribes the process by which a system can be designed to satisfy task related requirements, as they are captured by a representation of users' tasks. The discussion below identifies some open research questions, but also, it proposes a practical approach to using formal specifications in the design of user interfaces.

7.4.1 Some elements of the TKS theory

The TKS theory [108, 109] suggests that the knowledge a person has about a task is structured and stored in the person's memory as a *Task Knowledge Structure*. This knowledge is activated and processed during task execution and its structure is reflected in the behaviour of the person during task performance. A model of this knowledge structure, also called TKS, can be constructed by means of a task analysis. A TKS model consists in a *goal structure*, a *procedure set* and an *object structure*. These concepts are introduced briefly in the following paragraph. The interested reader is referred to [109] for an extended discussion of the psychological foundations of TKS, and to [185] for a comparison with some other task analytical approaches. As a psychological theory TKS focuses on the categorisation of the elements of user knowledge associated with a task, rather than a prediction of the phenomena that take

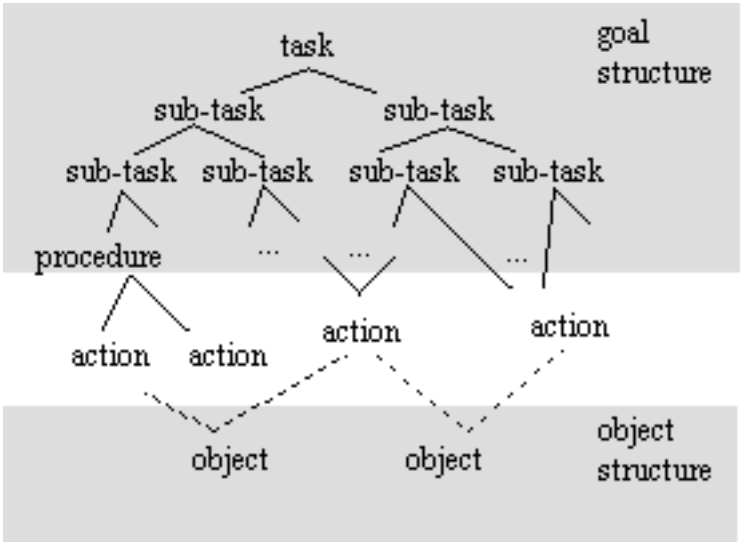


Figure 7.4. The main elements of TKS. Single lines indicate decomposition of task activity. Dashed lines indicate that an action is applied to an object.

place during task performance or during interaction (as for example is the case with the interactive cognitive subsystems theory [13]).

Consider a person engaged in a purposeful activity within a given domain. This domain is characterised by a state and the person is aware of a meaningful state to be achieved which constitutes that person's *goal*. In the process of achieving a goal, there may be identifiable, meaningful intermediate states that constitute *subgoals*. This decomposition of goals is structured into what is called the *goal structure*. This goal structure contains temporal information, which is important for the design of the user interface and which is explicitly represented in the formal task model of the next paragraph. The most basic element of activity identified in task performance is a single *action*. Actions are atomic, and they can be combined into subgoals. *Procedures* encode well rehearsed chunks of activity that are executed under appropriate conditions of the task domain. Actions apply to *objects*. Objects embody declarative knowledge of the task domain and are described in terms of their attributes and relationships to other objects. A TKS model can also incorporate indications of how necessary an object or an action is for a task, how typical it is for the task or even the frequency of certain actions. Figure 7.4 illustrates the decomposition hierarchy of the goal structure of TKS and its relation to the object structure through the task actions. The goal structure also describes logical and temporal relations between task components, e.g. the user must perform either of two sub-tasks, or both, an action may be optional, the tasks must be performed in sequence, or in any order, etc.

7.4.2 Formal representation of the temporal structure of a task

The temporal relations between task activities can be described by the standard LOTOS process algebra operators. The LOTOS operators and their meaning in the context of

Operator	Meaning in terms of the task model
<action>; B	Perform <action> then perform task B.
[b] -> B	If the Boolean expression b is true then perform B.
A [] B	Perform A or B
A ||| B	Tasks A and B are independent of one another, but can be performed simultaneously: they are multi-threaded
A >> B	Completing task A enables task B.
A [> B	Perform A until its termination, unless at some point B has to be performed thereby interrupting A.
A|[a,b,...]|B	Tasks A and B synchronise over actions a, b,.... For example, A may trigger B through action a.

Table 7.5. The meaning of LOTOS operators in the context of task modelling.

task specification are summarised in table 7.5. In [128] this use of LOTOS was introduced and some limitations of LOTOS for specifying the TKS goal structure were noted. LOTOS, as most formal languages, is obscure to the untrained and so difficulties arise regarding the validation of the model by users. As a task modelling notation it would benefit from some 'syntactic sugaring'. For example, [120] identifies the need for notation to represent the relationship between a set of subtasks that all have to be executed and completed without interruption, but where there is no preference or constraint as to the order of their execution. This relation between goals can be specified in LOTOS but not in a concise form. Further, LOTOS does not model true concurrency which can be useful to represent parallel tasks [108, pp. 174]. On the other hand, LOTOS is a powerful and standard notation with a well defined theoretical framework for reasoning about specifications and tool support for their manipulation. Tools support the symbolic simulation of specifications, which can assist the validation of the task model, their verification, and their testing.

The hierarchical representation of the goal structure, as in figure 7.4, can be extended with special nodes indicating the temporal ordering between the subgoals and actions of the task, giving a simple graphical notation for the task specification. The interested reader is referred to [127] where such a notation is introduced. A variant of this notation is supported by the Adept toolset [111]. More recently, Paternó and Mezzanotte [152] have used a similar graphical and formal representation of task knowledge for the TLIM method. The formal representation of TKS described here, can be used in the same way as the TLIM task model, with the difference that the components of the task model are derived by an explicitly defined method for the knowledge analysis of tasks [109]. The framework for relating the interface and task models which is discussed in the following paragraph, does not in itself require the use of the TKS model. Alternative descriptions of the task decomposition and temporal ordering could be used in a similar way. However the TKS model provides a clear link with task analysis and a psychological basis for the entities described.

7.4.3 Task based design and the property of task conformance

The requirement that an interactive system should be designed with due consideration of the user tasks has been called *task conformance.* Abowd et al. [2] propose a structured classification of properties for the principled design of interactive systems. According to [2] task conformance pertains to two questions:

- Task-completeness: Does the system address all of the tasks of interest?
- Task-adequacy: Does it support the users' tasks as the user understands them?

This description gives an indication of the issues involved but is rather ill-defined. While there are several theories as to how users understand their tasks there has not yet been a clear and widely accepted statement of principles concerning the mapping of task models to interface models [189]. This issue was raised in [122], where it was suggested that the relationship between task and interface model needs to be explicitly defined. In order to reach a formulation of task conformance it is instructive to examine a family of design approaches, which advocate the prescriptive use of task knowledge representations in the design of interactive systems. These are called *task based design approaches*.

Task based design emphasises the importance of understanding users' current tasks, the requirements for changing those tasks and the consequences that a design can have on a task [189]. Task models provide the focus for generating designs and help ensure that novel design ideas are motivated by a user-task perspective. Task based design approaches, e.g. CLG [135], TMM [89], Adept [188], ETAG [79], MUSE [114], and TLIM [153], etc., prescribe a design process by a set of models, their content, and notations for their representation. Figure 7.5 outlines the process of task based design as a progression between the models it involves. A task analysis of user task knowledge prior to the creation of the system design produces a model called the *current task model*. This model is the starting point of the design process. In task based design the task is redesigned and the result of this design activity is called the *envisioned task model*. This model describes how work tasks could be achieved with the introduction of the designed system rather than the operation of the system as such. The design proceeds with a specification of the interface to support these tasks, the *interface model.*

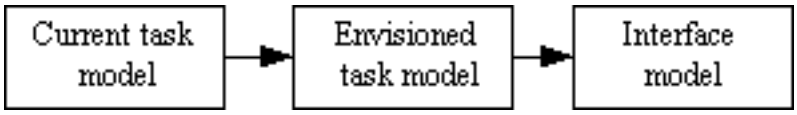


Figure 7.5. Overview of task based design (adapted from [189]).

This is a simplistic description of task based design which does not portray the structure of the models, their evaluation and the iterative nature of interface design. However, it helps describe some of the issues involved in task based design and it was for this purpose that it was introduced in [189]. Most task based design approaches mentioned are refinements of the process of figure 7.5, although not

all distinguish so clearly between current and envisioned tasks. Most do not describe explicitly how to progress from one model to the other. Wilson and Johnson [189] attempt to draw out and make explicit ways of using three types of knowledge represented in a task model to progress from the current task model to the interface model: the structural properties of user task knowledge, knowledge of task actions and objects, and the task sequencing knowledge. Their main recommendations are summarised below.

1. The structural knowledge of the user, i.e. the decomposition of a higher level tasks to lower level task components must be reflected in the structure of the user interface. The structure of the user interface pertains to 'groupings' of interactors. Components of the interface model that correspond to closely related components of the task model (goals, subgoals, procedures, actions, objects) should be grouped together in the user interface display. Grouping should be strongest at the lowest level of task activity, i.e. the actions associated with the same task component should be closely related in the interface.

2. Actions in the task model should be mapped to commands that the user will issue to the system. Additionally, objects suggest the types of information that may be manipulated by the commands. Therefore task objects and task actions may be directly supported by interactors. Complex objects may be supported by groups of interactors. The user interface design should support objects and actions to the user at an abstraction level determined by the task model.

3. The interface model should not violate task sequencing knowledge, i.e. it should not force the users to perform their tasks in a different order than that of the task model. This sequencing may be relaxed but the grouping of the interface components should still reflect the task model.

There are more possibilities for using task models to inform user interface design. For example, interaction tasks may be associated with some costing, leading to predictions of performance or mental workload as, for example, with the GOMS approach [32]. Relating task sequencing to a theory of errors may enable the designer to foresee how an interface design may be prone to erroneous interaction sequences, e.g. [71].

7.4.4 Relating task and interface representations

The ADC interactor model and the TKS formal model are expressed in the same formal framework but they are concerned with entirely different domains. Mappings between entities in the task model and those of the interface model are drawn using a conceptual framework which relates the two models.

The task model is by definition an abstract representation of user task related knowledge. Task actions determine an abstraction level at which the task can be related directly to the interface model. The assessment of an interface with respect to a task must abstract away from entities and behaviours related to lower levels of abstraction. In the

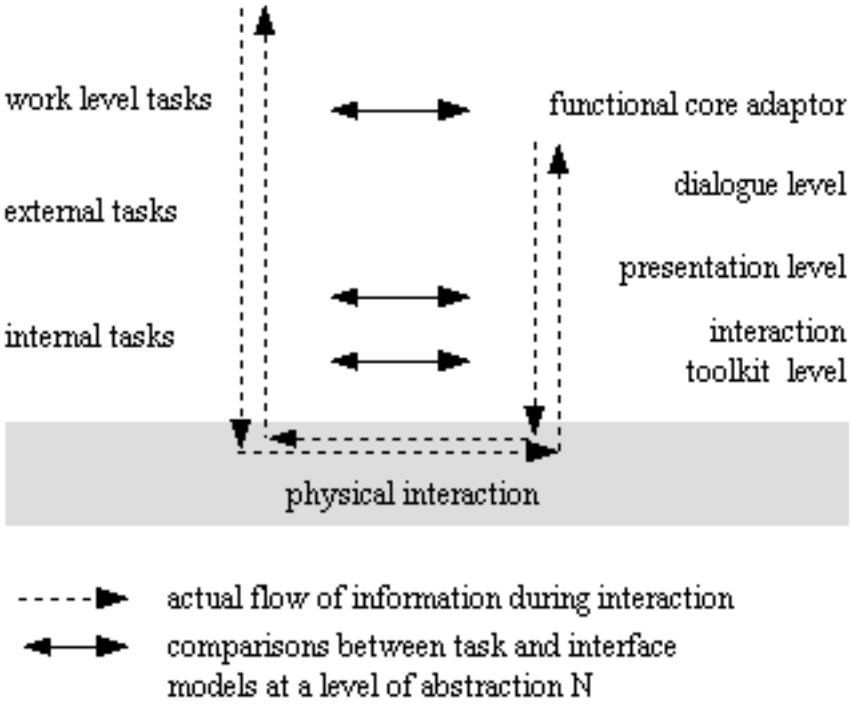


Figure 7.6. A framework for relating task and interface representations. Mappings between them at a given level of abstraction assume the operation of lower level entities.

framework of the ADC interactor model, this is achieved by studying interactors of the appropriate levels of abstraction, i.e. whose input and output actions correspond to task actions directly. For example, the actions of an interactor modelling the cursor of a text editing program are too low level to correspond to actions of the task of writing an article. On the contrary, the task action of setting the volume on a multimedia application can be mapped to the interactions with the volume control interactor of section 7.3.

Task knowledge may describe the users' work at a macroscopic level, which can involve many computer systems and persons and it can span over a long period of time. For example it may describe how to organise a meeting, to control the flow of airborne traffic over a given geographic area, etc. Task knowledge may also concern a small segment of activity, even the operation of a machine through physical actions, e.g. selecting an object on a drawing package, operating an automatic teller machine, etc. A point that has been stressed repeatedly throughout this thesis is that, similarly, the concept of an interactor applies to a wide range of abstraction levels. It may be the means to effect application functions, e.g. printing a document, changing the instructions to a pilot in an air traffic control system. It may also be an abstraction of a device, mapping physical actions to logical commands, e.g. keyboard, mouse, etc.

The general idea of relating tasks to interface models is illustrated in figure 7.6. Abstraction levels for interactors range over the components of the Arch model reference model. Tasks which model the intentions of the user are characterised as *external tasks*.

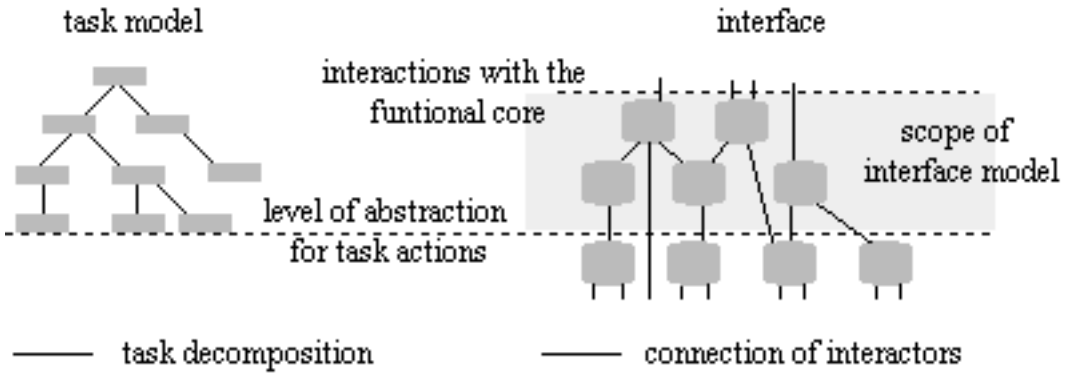


Figure 7.7. Drawing mappings between task and interface models.

Internal tasks describe how external tasks are achieved by using the system [108, pp. 6]. There does not need to be a one-to-one correspondence between the abstraction levels indicated for tasks and interactors. At the lowest abstraction level which describes physical interaction, perception and elementary motor movements by the user can be related directly to interaction with physical input devices. For example, studies of physical interaction by Card et al. [31] and Accott et al. [5] deal with this level. At the highest level of abstraction for the user interface, individual actions refer to abstractions of application functionality. Task descriptions may extend to tasks of the *work domain*, i.e. the world in which work originates, is performed and has its consequences [50]. In this world the user uses a computer to solve problems, although not all the tasks need to be directly supported by the computer.

Links between the task and the interface models can be drawn at any such abstraction level. If a task and an interface model are studied at an abstraction level N, it is implicitly assumed, that the interaction required to carry out the task is achieved via the immediately lower level of interaction N-1. For example, selecting a frame from a video sequence, may be considered as a basic action for the task of writing an article, for which the image obtained is used as an illustration. The details of how it is achieved, e.g. in terms of viewing a movie frame by frame and invoking the selection function for a frame, are not explicitly described in the corresponding task model.

Figure 7.6 does not imply a particular design strategy nor does it prescribe a standard set of abstraction levels through which the design should proceed in a top down fashion (cf. CLG [135] or the TMM method [89]). A similarly coupled view of user and system model is proposed by Barnard and Harrison in [12]. Contrary to their ‘interaction framework’, the framework of figure 7.6 does not aim to model the course of interaction or the psychological phenomena taking place as it unfolds. It is simply a conceptual aid for mapping task related requirements to the interface specification.

The discussion below focuses on the preservation of task sequencing information between the task and the interface model. The formalisation proposed describes a partial requirement in the transition from the envisioned task model to the interface model.

The level of abstraction at which the comparisons are drawn restricts the study of the interface model to a subset of the interactors, only those of a higher level of abstraction, and a subset of the interaction gates, only those that support input/output and synchronisation with interactors of lower levels. This idea is illustrated in figure 7.7. Interactions on other gates can be considered as 'internal detail' which is not directly of concern for the comparison with the task. For example, if the interface is modelled at the interaction toolkit level, input refers to pressing the mouse buttons, moving the mouse, etc., and the output refers to the appearance of interaction toolkit components, e.g. highlighting a menu item, checking a check box, etc. At a higher level of abstraction, the output might be to display a document, to output a set of values, etc. If the interface is described as a composition of interactors, only interactors of level of abstraction higher than of the selected set of gates need to be considered. Let G be the set of gates over which the interactions with the interface are observed. The interface is modelled by the following expression:

$$IM_G := \text{hide all but G in IM} \qquad 7.27$$

Some interactions may correspond to the same task action. For example, in the case study of chapter 5 there are many alternative ways to start Simple Player™, e.g. via the play/pause button, double clicking on the display, etc. All these alternatives must be renamed to indicate their correspondence to task actions, with a renaming $R_I: G \mapsto L$. The interface model becomes $IM_R = IM_G[R_I(G)]$.

Let A be the set of all task actions that have a direct correspondence to interface actions. The correspondence of task actions to the actions of the interface model can be represented by a mapping $R_T: A \mapsto L$. Some task actions may not correspond directly to an interaction. For example, they may represent user decisions or simply task actions which are not supported by the interface, e.g. telephone communications, etc. The interface model should not support corresponding interactions but these actions are significant in describing task sequencing.

There may be gates in G that do not correspond to a task action, e.g. representing tasks actions introduced in order to control the user interface. A different task requirement results if these interactions are modelled explicitly or not. If an idealised description of the task is adopted which does not include these actions, a comparison with the interface has to consider the relevant interactions as internal detail. The comparison of the interface to the task specifications will reveal the feasibility of performing task actions in the specified sequencing but ignoring 'interaction tasks', e.g. bringing the system to the appropriate mode, house-keeping of the interface, etc. In the discussion that follows, it is assumed that the envisioned task model has been extended to incorporate internal tasks necessary to interact with the interface modelled by IM_G, so the mapping R_T may be considered surjective. The task model describes what is an acceptable interaction with the system to perform the required tasks, and the comparison of the interface model to the task model portrays the fit of the interface to this task description. The task model is renamed according to the mapping R_T, so it is represented as $TM_R = TM[R_T(A)]$. The interface model IM_G and the task TM_R are both modelled as LOTOS processes or, more

generally, as labelled transition systems, so the next question that arises is how to compare them formally.

7.4.5 A formal definition of task conformance

The choice of the mathematical relationship required between the two models should reflect on how a human observes a computer system during task performance and how the human compares observed behaviours. Defining what is an observation and how its outcome may help distinguish or identify systems is a hard problem, even for the traditional applications of formal methods that are not concerned with human cognition and behaviour. Various researchers, e.g. Brinksma [29] and de Nicola [45, 47], discuss alternative models of observing and comparing of system behaviours. Each method of comparison offers different 'discriminating power' between system specifications. The appropriateness of these concepts varies with their intended use.

In chapter 6, bisimulation equivalence and some of its variants, e.g. strong, weak, φ−bisimulation, etc., were used to relate the input and the output forms of transformations (their definitions can be found in appendix A.1). This was appropriate in that context to establish the strongest (useful) relation between the input and the output forms of the transformations. Paternó [148] compares user interface specifications with respect to observational equivalence [133], comparing the specified interaction with the functional core or with the user. However, for reasons discussed extensively in [45], observational equivalence is too strong a requirement for comparing user interface systems. It discriminates systems which can not be distinguished on the basis of their interactions with external stimuli. A weaker comparison is to compare just the traces of a process. In the context of comparing two interfaces, this means that two interfaces that offer the same sequences of actions starting from their initial state, but offer different options as the interaction unfolds will be considered equivalent. Clearly, the comparison of interfaces should use a more discriminating method than the comparison of traces and a less discriminating method than observational equivalence.

A comprehensive review of the various proposals for comparing system behaviours and a cognitive theory of how users perceive and compare interface behaviours are outside the scope of this section. The reader is referred to [45, 10] for an extensive treatment of the topic. Another important issue is to choose carefully which actions to consider in comparing behaviours and how they are interpreted with respect to the actual interaction phenomena. For example, if output actions are modelled, is it assumed that the user always perceives what is output by the system?

The formal concept of *conformance* [27] is proposed here as a suitable relation for comparing interface to task model, which embodies the intuitions regarding the preservation of sequencing information during task based design. A formal framework for comparing interactor specifications with respect to their responses to finite deterministic tests is summarised below. The reader is referred to [117] for a more comprehensive presentation of this testing theory.

Definition. Conformance of behaviour expressions.

Let Q_1 and Q_2 be processes and let L be the set of all possible labels for all LTSs.

Q_1 conf Q_2 if

$$\sigma \in Tr(Q_2) \cap A \subseteq \mathcal{L} \cdot$$
$$\text{if } Q_1' \text{ after } \sigma \text{ and } A \cdot Q_1 \text{ refuses } A \text{ then } Q_2' \text{ after } \sigma \text{ and } A \cdot Q_2 \text{ refuses } A \qquad 7.28$$

If Q_1 can perform some trace σ and then behave like a process Q_1' and if Q_2 can perform the same trace σ and then behave like Q_2', then the following conditions are required: whenever Q_1' refuses to perform an action α from a set A then Q_2' must also refuse every action in A. In other words, Q_1 conf Q_2 means that testing the traces of Q_2 against the process Q_1 will not lead to deadlocks that could not occur with the same test performed with Q_2 itself.

A formal expression for task conformance can now be written as follows:

$$IM_R \text{ conf } TM_R \qquad 7.29$$

This expression means that a user interacting with an interface, that behaves as IM, will not reach an impasse when performing a task, as described in TM, when task actions in A can be related to interactions on gates G of the interface, via two mappings R_I and R_T as in the previous section.

IM_R may specify behaviours which are not specified in TM_R, in other words, the task model is a partial specification of requirements for the interface model. Conformance is an appropriate relation for comparing the task and interface models, because it is not symmetrical and also because it is sensitive to the non-determinism that results from hiding the internal behaviour of the user interface. What is primarily required from the interface model is that all tasks specified in the task model are possible. Therefore the interface should conform to the behaviour specified by the task model and the mapping R. However, it is not required that an interface model be limited to the tasks described by the task model. Conformance can be verified through a comprehensive deadlock analysis, but perhaps more interesting for practical purposes it can be tested [27]. A set of tests can be constructed from the task model and they can be applied to the formal specification of the interface or to the actual interface software without knowledge of its internal structure.

Testing compares systems with respect to their response to a set of finite sequences of interactions with the environment, the tests. A formal definition of conformance, due to Brinksma [27], which is based on testing is summarised below.

- A *test suite* is a set of processes which are called *test cases*.
- Let $\sigma \in A^*$ and T be a test case. A derivation $T||Q \xrightarrow{\sigma} T'||Q'$ is a *test run* of T and Q. A test run is *completed* when $T'||Q' \sim stop$.

- A completed test run is *successful* if its last event prior to terminating is a reserved event *success* signifying successful termination. A completed test run *fails* if it is not successful.
- A test case T is successful, denoted as Succ(T,Q), if all test runs of T and Q are successful. A test suite is successful if all its test cases are successful.
- Let S be a process and $\mathcal{Q}$ be the set of all possible processes. The *canonical tester* of S is a process T(S) such that

$$Tr(T(S))=Tr(S) \quad Q \in \mathcal{Q} \mid Q \text{ conf } S \text{ iff } Succ(T(S),Q) \qquad 7.30$$

It has been shown [27] that for all LOTOS process specifications there exists a canonical tester. The details of the generation of the canonical tester are not given here. Various algorithms and tools have been proposed to solve the problem for test generation, e.g. [117, 69, and 184]. The generation of the canonical tester is supported by LOTOS general purpose tools [30, 119].

Verification of task equivalence

An interesting property of the canonical tester is that it relates testing and verification. In [27] it is shown that by the definition of the canonical testers it follows that Q_1 conf Q_2 if every deadlock of $T(Q_2) \parallel Q_1$ can be explained by $T(Q_2)$ having reached a terminal state, i.e.

Q_1 conf Q_2 if

$$\sigma \in Tr(Q_2) \mid \alpha \in A \cdot \text{if } T(Q_2) \parallel Q_1 \xrightarrow{\sigma} T(Q_2') \parallel Q_1' \xrightarrow{\alpha} \text{ then } T(Q_2) \xrightarrow{\sigma} T(Q_2') \xrightarrow{\alpha} \qquad 7.31$$

Thus task conformance can be verified by a deadlock analysis on the parallel behaviour expression $T(TM_R) \parallel IM_G$. If there is a trace leading to a deadlock of this expression, but not for $T(TM_R)$ on its own, then the interface model is not conformant to the task model. If all deadlocks of the synchronous composition $T(TM_R) \parallel IM_G$ are also deadlocks for TM_R alone, then the interface model conforms to the task model. Deadlock analysis can be carried out by the Caesar/Aldebaran toolset [68]. The Aldebaran tool may be used to detect deadlocks of an LTS, also providing as a diagnostic the trace that leads to the deadlock. However, this may not be desired because of the size of the models which may be prohibitive, or because it is preferred to test the actual software rather than its formal specification. In practice, it is unwieldy to compare the sets of the traces that lead to deadlocks for two specifications, as this comparison is not directly supported by the general purpose tools for LOTOS and it has to be done by hand.

A practical method for testing task conformance

An alternative to comprehensive deadlock analysis is to generate a finite set of tests that give a reasonable coverage of the behaviour of the canonical tester. These can be tested against the interface specification on a specification test bed such as the LOLA tool [156]. Also the tests, or rather the implementations of these tests, can be applied directly to the actual interface system. Note, that in this case passing a test does not mean that the interface software or its formal model actually conforms to the task model, since the interface may be non-deterministic. Each successful test case is an indication (only) that the system is conformant. A failed test case means that the system is not conformant to the task.

A practical method of testing the task conformance of an interface model can now be outlined:

1. Select a level of abstraction for studying the task in question and define the interface model as the composition expression involving interactors of a higher abstraction level. Determine the set of gates G through which lower level interactors communicate with the selected interactors. Define the correspondence of task actions to interface actions by the mappings R_T and R_I. These mappings are specified through the renaming of the gates of the specifications TM and IM.

2. Produce the 'canonical tester' $T(TM_R)$, e.g. using a test generation tool like COOPER [30, 184]. Let $CT = T(TM_R)$. Hide all gates of the task model which do not correspond to interactions with the user interface.

3. From CT produce a set of finite tests T_i, with i:1..n, that the interface specification will be tested against.

4. Run the tests against IM_G. Testing may be performed on a test bed tool, like the LOLA component of the TOPO toolset [156]. A full exploration of the synchronous composition of a test t_i against the interface model should give a 'may' outcome after a full exploration of all behaviours.

Conformance is not a pre-order relationship [29]. This means that when an interface specification IM_1 conforms to a task model TM, and a more detailed model of the interface IM_2 is specified that conforms to IM_1, then IM_2 does not necessarily conform to TM. Testing is promising as a practical framework for the validation of an interactive system with respect to task requirements. A test suite can be expanded and refined with implementation constructs during the development of an interactive system. A realisation of a formally specified test suite may be used to test the realisation of the system, possibly with the involvement of users. Testing can thus link the formal specification of interfaces and tasks with other stages of the development of the user interface. A small set of tests of finite size may reveal interesting problems with an interface design and can focus the attention of the designer on the behaviours required by the task model. In comparison, model checking requires the generation and manipulation of quite sizeable models even when very small specifications are

concerned. A practical question that arises, is how to choose an effective set of tests that provides a good coverage of the task behaviour and is economical as well. This issue is a subject of future work and is not addressed here.

Interactions which do not relate directly to task actions are a source of design concern. At one level they may be abstracted away from, encouraging the current and the envisioned task models to be described independent of the tools that support the task. The corresponding gates can then be considered internal detail of the interface model for the comparison with the task. In this case, conformance pertains to whether task sequencing is preserved without respect to intermediary interactions, e.g. changing modes, navigating through screens, etc., which are crucial to describe the interaction tasks. When such interactions are explicitly modelled in the task model the conformance relationship reveals differences in the way the interaction dialogue hinders or supports the task sequencing described in the task model.

7.4.6 Example: Testing for task conformance.

The remainder of this section demonstrates how a model of a task may help assess two interface designs. The task is fabricated for the purposes of the discussion. It is envisaged that this type of assessment should follow a task analysis. The exemplar task is to produce a small text, e.g. an invitation to a party, which has already been composed with a word processor. The imaginary user experiments with the typesetting, i.e. changing the fonts, letter sizes, column layouts, margins, etc. The task is performed with the Microsoft Word application. The example compares how versions 5.1 and 6.0 of Microsoft Word fare with respect to the particular task. Both versions have the same functionality as far as this task is concerned but their interfaces are different. In reality, the two versions are distinct applications whose functional core offers different functionality. However, this difference can be overlooked here since the functionality accessed through the two interfaces is the same.

For brevity, the task description is restricted to changing the orientation of the page, which is effected through the ‘page set-up’ dialogue box of Word, and changing the margins, which is effected through the ‘document layout’ dialogue box or by direct manipulation on the ‘print preview’. Print preview is an operating mode for the word processor, in which the text is displayed to resemble its appearance in printed form. In this mode the user cannot modify the text content of the document, as is possible in the standard mode of a word processor, (the ‘Normal View’ or ‘Page Layout View’ of Microsoft Word). Unlike Word 5, Word 6 lets the user set the margins and the orientation of the document in the preview mode as well as in the standard mode.

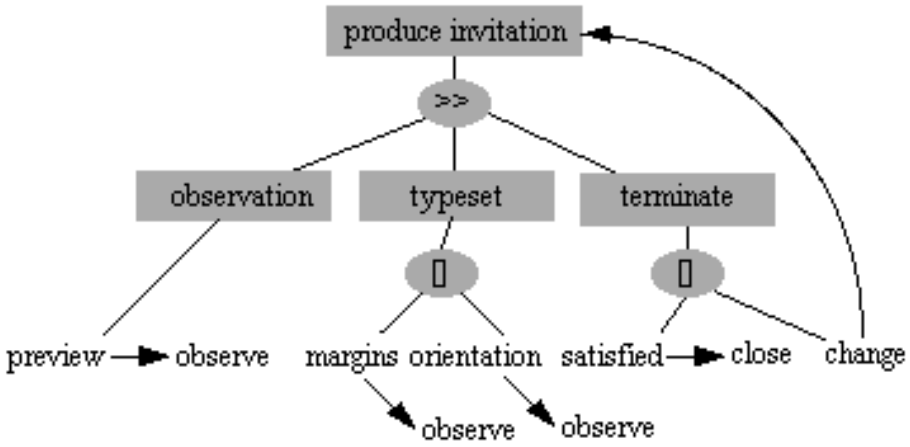


Figure 7.8. Diagrammatic illustration of the task 'produce invitation'.

Figure 7.8 illustrates the goal structure of this simple task, annotated with LOTOS operators to indicate their temporal ordering. The task *produceInvitation* is defined with three subtasks *observation*, *typeset* and *terminate*, which are performed in sequence (operator >>). The subtask *observation* involves the interaction *preview* and the task action *observe* performed in sequence (the arrow between the two actions stands for the action prefix operator). Typesetting is described here as a choice of two actions setting the margins and setting the orientation. The subtask *termination* is where the user decides whether to make more changes, in which case the task is repeated. If the user decides that the result is satisfactory then the task terminates. The LOTOS specification of the goal structure is as follows:

```
process  produceInvitation[...]: exit:=
    observation[preview, observe]  >>
    typeset[margins, orientation, observe] >>
    terminate[margins, orientation, preview, observe, satisfied, close, decideAChange]
endproc
process observation[preview, observe] : exit :=
    preview; observe; exit
endproc
process typeset[margins, orientation, observe] : exit :=
    margins; observe; exit
[]  orientation; observe; exit
endproc
process terminate[margins, orientation, preview, observe, satisfied, close, decideAChange]:exit:=
    decideAChange;  produceInvitation[...]
[]  satisfied; close;    exit
endproc
```

The task is defined recursively, so possibly infinite cycles of activity follow from the above description. The task action *observe* is mapped to interactions on gate *showPrv*. The actions *satisfied* and *change* represent user decisions and have no image on the interface model. A set of finite tests were derived (manually) with the process described in the previous paragraph. Each is a finite sequence of steps ending with a success

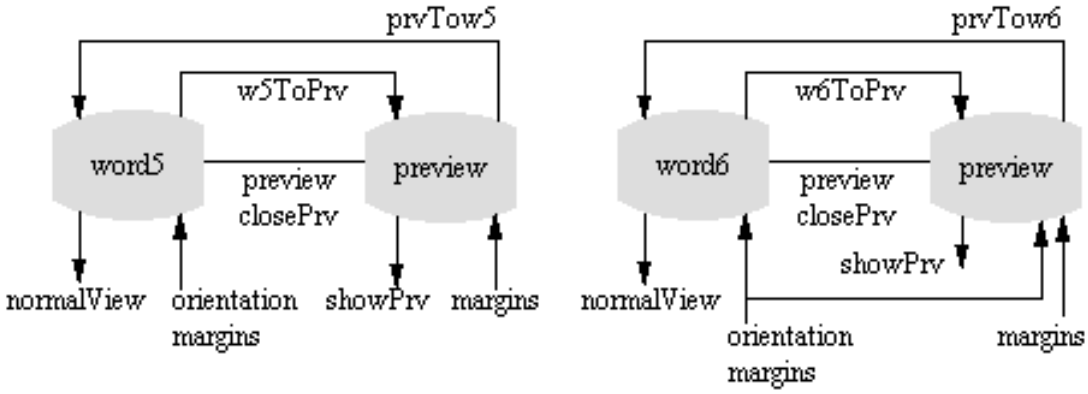


Figure 7.9. The specification architecture for Word 5 and Word 6. Word 6 supports interaction to set the margins and the orientation, even through the preview interactor.

event, to signify the successful termination of a test. For example, the tests t1 and t2 below have been derived from the canonical tester.

```
process t1[preview, observe, margins satisfied, close, success]: exit :=
    preview; observe; margins; observe; satisfied; close; success; exit
endproc
process t2[preview, observe, orientation, satisfied, close, success]: exit :=
    preview; observe; orientation; observe; satisfied; close; success; exit
endproc
```

The test *t1* describes a task action sequence where the user observes the preview display, sets the margins in any of the two ways supported, checks the result of this action, is satisfied by it and terminates the task. The task action satisfied must be hidden so the actual test for the interface model is:

```
hide satisfied in t1
```

For both versions, the interface is modelled as the parallel composition of two interactors. Interactor *word5* (respectively *word6*) models the standard interface to the word processor during editing operations. In the particular word processor, this may be either the ‘page layout view’ or the ‘normal view’ of the edited document. In both versions, the final result can be inspected through the preview facility, which is modelled by the interactor *preview5* (respectively *preview6*). The difference between the two versions is that in Word 6 the document layout and page orientation dialogue boxes, can also be accessed through the preview interactor, along with all editing operations that apply to the whole document.

The interactor labelled *word5* in figure 7.9 may invoke the interactor *preview5*. This is achieved via a control gate that is connected to the *start* and *resume* (formal) gates for the latter. *Word5* also sends data to *preview5* through gate *w5ToPrv*. Changes to the document effected via the *preview5* interactor, are communicated to *word5* through the gate *prvToW5*. The suspend gate of the *preview5* interactor is connected to a control gate of *word5*. The specifications of *word6* and *preview6* are very similar. The only difference is that *preview6* receives also orientation information from the ‘page set up’

dialogue box. Relevant changes will appear in the data type specifications of *preview6* and the controller component.

Testing *t1* against version 5 shows that it may succeed while *t2* is rejected, indicating that version 5 does not conform to the task *produceInvitation*. The result is 'may succeed' for both tests with version 6. This is a positive indication for task conformance, although not definitive, given that the test suite is only of a finite size.

7.4.7 Related work

The concept of a task-template [160, 161], discussed briefly in section 3.5, was an attempt to relate system models to models of user's tasks. Task-templates can be thought of as 'filters' that select those elements of a system specification which are necessary to perform a task. In association with the state-display model they provide a framework for an analytical use of task representations. The 'syndetic' model of [63] uses interactors to model an interactive system and also as a representation of a user, informed by the Interactive Cognitive Subsystems theory [11]. This syndetic model provides a formal framework for analysing the cognitive component of interaction in terms of the mental resources needed to use a device for a specific task.

In contrast to these analytical and theoretically motivated approaches, [147] draws parallels between two formal methods for design, which relate representations of user tasks with system behaviour. These two methods are called TLIM and MICO. TLIM [153] uses a formal specification of the user's tasks in the LOTOS formal specification language. The task model specifies the temporal relationships between component tasks using LOTOS, in a very similar fashion to the formal representation of TKS described previously. The system is modelled as a composition of interactors using the interactor model of [150]. The task specification is mapped to the interactor-based description by means of an algorithm which aims to ensure the compliance of the system to the task constraints. TLIM generates an architectural design from the task specification so the task model may be seen as a very abstract system specification.

The MICO design method models tasks and systems using the Interactive Communicating Objects formalism ICO [145] which is based on Petri-Nets. MICO does not prescribe how the system and the user model are derived, but it suggests that the design of the system involves the iterative re-design of both system behaviour and user tasks. MICO supports this iterative design process, by providing a framework for specifying formally the user interface and the user tasks. The two models are merged into a single representation, which is analysed to determine whether the system conforms to the task. The 'semantic consistency' of task and interface model is verified in the MICO method [146] by a deadlock analysis of the combined task and interface representations. This corresponds to deadlock analysis on the expression $TM_R \parallel IM_G$, which verifies the feasibility of completing the task rather than that task sequencing is supported by the interface model.

The formal framework proposed in this section shares some characteristics with the related approaches mentioned. The ADC interactor model is similar to that of the [150], so the method discussed in this section could extend also to that interactor model. Like the syndetic model of [63] it adopts a psychologically informed method for describing aspects of user cognition. However, it does not do so with the aim to provide a model of the interaction of system and user, nor does it use the task model as an abstract specification of the system. Rather, as is the case with the MICO method, the two models are developed independently. The formal framework enables their combined analysis to ensure the conformance of the user interface design to a given task model. The approach presented in this section places its emphasis on defining this notion of conformance. This definition is informed by the consideration of task based design approaches, which try to ensure the conformance of an interface design to a task by process, rather than by an explicit formulation of the conformance relationship.

Paternó et al. [155] report the combined use of interactor models and task models in order to evaluate an interface design. They describe a prototype system for logging user interactions and mapping these logs to user tasks via mappings derived from the user interface software architecture. While it has different objectives, the research of [155] is further testimony that models of interactors and tasks can play a bridging role throughout different stages of the design activity.

7.4.8 Discussion

The discussion on task based design has focused on how it supports a progression from a task model to an interface model, and in particular on how task based design aims to support task sequencing. The definition of the task conformance requirement is determined by two main hypotheses:

1. The formal notion of conformance captures the required relationship between task and interface model. Few attempts have been made to explicitly define this relationship. A rare example, is the MICO method [146, 147], mentioned above, which seeks to establish deadlock freedom for the combined task and interface behaviours.

2. The task and the interface models are compared at the level of abstraction determined by the task model, which in turn is defined by the task analysis. Interactors of lower abstraction levels are ignored and behaviours internal to the interface model are abstracted away (as illustrated in figure 7.7). Similarly, the relationship between task and interface model relates explicitly task and interface actions. All other task actions are a source of non-determinism as far as the system is concerned, e.g. the user makes a commitment to a particular course of action which is not ‘known’ to the interface.

The set of tests produced from the task model may be used to test different interface designs and at varying levels of abstraction. Testing in this sense may be associated with techniques from the domain of human computer interaction. A suite of tests derived

from the task model may benefit the evaluation of user interface designs, in combination with systematic evaluation techniques as, e.g. user testing [186], cooperative evaluation [134], etc., which rely on either users or designers working through a predefined set of tasks. The approach described hereby could help generate systematically the required suite of task action sequences from the task model. The addition of concrete detail to test cases is an interesting aspect of testing, e.g. specifying data exchanged with interaction, the information to be extracted from the display, upper limits on the response time, etc.

By adding concrete detail and contextual information about the situated work of the user, a test may be developed to a user interaction *scenario*. A scenario is a concrete description of what people do and experience as they perform a specific task using the designed computer system. Scenarios can be used in many ways during the development of an interactive system. Most relevant to this section is their use for evaluation. Nielsen [141] discusses the use of scenarios in combination with heuristic evaluation. Heuristic evaluation is a highly informal usability inspection technique, where a set of expert evaluators inspect a user interface in order to generate a list of usability problems in the interface. This inspection is normally not associated with a set of scenarios which can constrain the evaluator. However, to support the design of highly domain dependent interaction [141] recommend using a set of scenarios derived from a task analysis.

The approach presented has put a lot of emphasis on the use of tools to support the construction and validation of the models, their verification, etc. Tools support the generation of tests only in part, mainly because from the canonical tester it is possible to derive an infinitely large set of tests. An important practical question is how to select the most interesting tests, and having performed some tests how to use their results to determine which other tests to perform. From the point of view of task based design it is most important to establish principles to help prescribe the mapping R between task and interface actions and to determine how the envisioned task model is designed from the current task model.

7.5 Conclusions

Several uses of the ADC model have been discussed in this chapter. They reflect on some of the main themes of formal methods research in the context of human-computer interaction. The purpose of the discussion has been to show the versatility of the ADC interactor model that enables different approaches to be integrated in a single conceptual and formal framework.

Section 7.1 revisited the concept of generative user engineering principles (GUEPS) for user interface design. This concept has influenced much of the early research work in formal aspects of human computer interaction. Formulations of GUEPS in terms of the ADC model were proposed in the framework of LTS. The expressions follow the general classification scheme of Sufrin and He [169], although slightly different interpretations of the relevant properties are proposed. The verification of this class of

properties poses problems mainly because of the size of the LTS representations of interface specifications. Rigorous reasoning seems to be a more viable alternative. Consequently, it seems that this class of properties are more appropriately defined and verified with more abstract, and therefore economical, specifications.

The verification of dialogue properties is a more tractable problem. In terms of expressive power and of the possibilities for verification of this class of properties, interactor specifications do not offer more expressive power than traditional dialogue representations, e.g. PPS [142], Statecharts [81], etc. They offer, though, a conceptual framework for the specification of these properties. The specification of dialogue properties is facilitated by the standard structure of the interactor. Rather than inspecting a monolithic dialogue model to define or verify properties, the expressions presented relate classes of actions easily identified with the interactor model. The expression of dialogue properties using a branching time temporal logic, and their verification using a general purpose model checking tool, constitute a workable verification method, introduced with the Pisa interactor model. The relevant expressions were easily reformulated in terms of the ADC interactor model. Further, it was argued that the compositionality property of the interactor model may help formulate dialogue properties for complex interfaces that are composed by many interactors. An alternative to the use of a temporal logic was illustrated, which is to specify dialogue properties in LOTOS and verify the equivalence of the interface specification with the required property specification.

In the final two sections the ADC interactor model was discussed as a design representation used within a top-down design of the user interface and in a task-based design approach. The examples presented were brief and only serve to illustrate the potential of the interactor model. As with the verification of dialogue properties the use of tool support is necessary for anything but trivial examples. Without such tool support, few claims can be made regarding the practical benefits that the ADC model offers in those contexts.

In all four sections, of this chapter a constant theme has been the use of established methods and tool support. The obvious advantages of this policy is that technology currently available is used and future developments of this technology will benefit this research. Apart from the verification of dialogue properties of a user interface, existing generic tool support has not met the requirements for the problems discussed. For example, the automatic verification of display predictability in section 7.1, requires some operations (simple graph search and comparison of nodes) not offered by the tool support discussed. As discussed in section 7.1 observability would require more sophisticated mappings between related LTS. Stepwise refinement was demonstrated through examples edited by hand. It was mentioned already that tools need to be developed to support the automatic transformation of ADC specifications. In the case of task conformance, verification relied on the comparison ‘by hand’ of the diagnostics given by the model checking tools. Even for test generation, the test cases must be written by hand to eliminate the recursion in the canonical tester. In some cases what is required is that current tool support be extended to accommodate the specific problems

discussed, e.g. for verification of predictability and observability properties, for test generation, etc. In other cases, like in the case of applying transformations it seems necessary to develop special purpose editing facilities.

In section 7.4 the discussion addressed a topic with important repercussions for all other sections. This is the choice of appropriate semantics to model the way users perceive and compare temporal behaviours, e.g. bisimulation, trace, failures, etc. Testing of input and output of an interactive system was suggested as a reasonable approximation, but clearly the choice of the appropriate semantics is a problem for research into the psychology of the user. It seems to be an interesting research question, but as was mentioned already, the answer depends also on what are the elementary actions for the user and machine, so potential answers are particular to the system and user models studied.

By adopting testing as a working approximation for comparing interface specifications, a practical approach emerges for the testing of interfaces. In section 7.4, testing was used as a means to establish the property of task conformance. Task conformance, as well as other properties of user interfaces, may be modelled by finite sets of tests, which can be progressively refined and updated and even tested against the realisation of the interface. Whether this is a workable technique in the pragmatic world of user interface development is still an open question. At least, the discussion of this section suggests an interesting avenue for further exploration, that promises to bridge the gap between theoretical models of human cognition, formal specifications of interface designs and the practical concerns of user interface development.

Chapter 8

Conclusions

This chapter summarises the thesis and puts forward a list of emerging research questions and some suggestions as to how future work can address them. The chapter rounds up the thesis with a brief assessment of its contributions.

8.1 Summary of the thesis

This thesis has investigated the application of formal methods to the development of user interface software. In particular, it has sought to establish appropriate abstractions for describing user interface software and a formal scheme for their representation.

The approach taken to this research question has been to draw lessons from two research fields which have proposed abstractions for user interface software. These are the fields of user interface software architectures and formal models of interactive systems. A standard specification language (LOTOS) was adopted for the specification of user interfaces, to benefit from its mature theoretical foundation and existing general purpose tool support. An important motivation for the thesis has been to develop a practical scheme for the specification of user interface software. Reusable templates for the specification of user interface software have been defined, which embody the general characteristics of the proposed abstractions and which support the systematic reuse of specification components and a compositional approach to writing specifications.

The thesis has focused on interactor models which are formal abstract representations of user interface software at a detailed construction level. An important consideration has been to map results and techniques originating with more abstract models of interaction (such as those discussed in chapter 3) to this concrete level. The Abstraction-Display-Controller (ADC) interactor model has been put forward as a formal abstraction that can fulfil the role described. The thesis has described the requirements for the model, has documented the properties of its formal representation and has discussed some of its potential uses.

The ADC functionality is described partly by the operations upon its abstraction and its display state components and partly in terms of the dialogue it supports. These two descriptions are linked by the architectural concept of a 'gate'. A gate groups interactions with a similar purpose. This can be to input or output data, to apply some operations on the state components or simply to synchronise with other interactors without any effect on the state parameters of the interactor. The temporal ordering of interactions on gates is described in the controller unit. The abstraction, the display, the operations upon them, the gates and the temporal ordering of interactions are orthogonal dimensions for describing user interface software. This orthogonality can facilitate the articulation and development of design decisions and the comparison of designs at an informal level. The designer can use the ADC interactor as a conceptual framework to construct a model of an interface and its components.

The ADC interactor has been specified formally as a template of a LOTOS process definition associated with an abstract data type. The compositionality of the model has been demonstrated by a set of theorems which describe transformations of LOTOS behaviour expressions. A case study in the use of the ADC model was reported, which tested the model and subsequently led to its improvement. The case study is a formal specification of considerable complexity which describes a real software system. Several uses of the model were discussed in chapter 7. This discussion stressed the integrating role of the ADC model which enables the expression and verification of properties of the user interface, in terms directly relevant to the interface architecture. The use of the model for the stepwise refinement of an interface design specification to its realisation was exemplified. Finally, the ADC formal interactor model was discussed in conjunction with a psychologically based model of the users' tasks to assess aspects of the usability of a user interface design.

8.2 Discussion and Future Work

This section discusses the ADC interactor model from a variety of perspectives, raising some issues for future research. The research agenda it provides is an important product of the thesis. Some of the research questions raised are currently the subject of investigation, as part of the ARAMIS (Applying Requirements on Architectural Models of Interactive Systems) research project at Queen Mary and Westfield College. The ADC interactor model has been adopted as the formal system model for this project.

Relevance of the ADC interactor model to implementation practices.

One of the early decisions of the reported research was that formal models of user interface software should resemble user interface architectures. In chapter 2 several user interface architectures were discussed and the ADC interactor model was shaped so that it embodies some of their characteristics, e.g. modularity, orthogonal dimensions for its description, etc. In particular, the compositionality of the ADC model is intended to

resemble the compositionality of some of these informally defined architectures, e.g. PAC [39], ALV [95]. An argument that can be levelled against this approach is that the research models discussed in chapter 2, have had to date only limited impact on practical user interface development. Presently, user interface development is based on the use of toolkits and the technique of call-back functions. This questions the appropriateness of a compositional model as a practical aid for the implementation of user interfaces. However, there are merits to supporting compositionality: it helps conceptualise an interface at various levels of abstraction, it helps structure an interface specification, it assists the analytical use of the specification. The relevance and usefulness of compositionality for developing user interface software remains to be tested with future applications of the model, but eventually it may well depend on developments in software platforms for implementing user interface systems.

The ADC model as a conceptual framework

This thesis has been concerned mostly with the formal representation of the interactor model, but the importance and the utility of an informal description should not be overlooked. There is a real need for less formal frameworks for describing user interfaces as, for example, the User Action Notation (UAN) of [88]. This notation does not have a formal syntax or a semantics, but there is some evidence that user interface designers prefer UAN over other more formal representations like temporal logic and Petri Nets [107]. A plausible explanation is that the scheme is easy to grasp and highlights salient aspects of user interface software. Compared to UAN, the ADC interactor model has the advantage of being modular and it has a direct mapping to a formal representation. The ADC model has not been used independently of its formal representation, but this use seems a worthwhile subject for future research. Such an investigation may help bridge the ‘formality gap’, discussed in chapter 3, between design requirements and their formal specification. Regarding the narrower objective of developing the ADC interactor model, the use of the ADC interactor model as a conceptual structure independently of its formal representation, will provide evidence of the utility and validity of the model.

Object-orientedness and the ADC interactor model

The investigation of user interface architectures, in chapter 2, has identified the concept of an object as an elementary architectural unit. Also, the concept of an interactor was introduced, in chapter 3, as a specialisation of the more general concept of an object, in the object oriented software engineering sense. The ADC interactor has been modelled formally as a process rather than as an object. It is worth comparing an ADC interactor to the general notion of an object and to assess whether this disparity influences the appropriateness of the ADC interactor for the specification of user interface systems.

Interactors and objects alike are encapsulated specification entities which are capable of interacting with similar entities. They each possess a set of operations and a state that

records the result of the operations. The state can be accessed externally only through a well-defined interface of the interactor (respectively the object) with its environment. It has been suggested [40, 164] that the following concepts characterise object-oriented specification languages:

- An *object*, considered as a unified notion of a module of specification. Objects have to be encapsulated, i.e. they interact with their environment through a prescribed interface. The language must support a standard model of communication.
- A *class*, considered as a collection of similar objects.
- *Inheritance*, considered as a programming/specification engineering technique by which class definitions can be modified incrementally.

The notion of *object based* specification, or programming, is used for a language that supports only the first of the concepts listed above. Clark [36] defines object based specification in LOTOS as a specification style. A close examination reveals that it is identical to the resource oriented specification style [180], described in chapter 3. This definition characterises the ADC interactor as object based.

However, the concept of an object is not fully supported by the ADC formal interactor model. Objects need to be created and destroyed dynamically at run-time, and this notion of the lifetime, or existence, of an object with its own identity, should be reflected in an object-based specification. An ADC formal interactor can describe the lifetime of one interactor for which the connections and the context for its execution are specified statically in advance. The ADC model does not readily generalise to an indefinite number of objects following the same specification or whose configuration changes dynamically. For this generalisation, the model must be extended so that starting an interactor does not just start a process, but creates an individually identifiable instance of the ADC interactor. Supporting the notion of an identity for the object implies also that all interactions need to be associated with object identifiers for sender and recipient, to distinguish between objects of the same class. These extensions are trivial to apply to the ADC model and its transformations although they would clutter the specifications. On the other hand, more effort is needed to define a scheme for managing dynamically the connections between objects, and to model their dynamic creation and destruction. This is a challenging task in the framework of LOTOS, which is intended to support the specification of static configurations. A specification ‘style’ for supporting object based specification in full LOTOS without the need for semantic extensions of the language has been reported in [140]. The combination of such a scheme with the ADC formal representation could be an interesting avenue to explore in the future.

Higher levels of object orientation, as characterised by Meyer [131, pp. 60], should support the concepts of object classes and inheritance. The ADC interactor model does not support inheritance, i.e. it does not provide syntactic operators to specify incrementally a class of interactors by modifying the description of its parent classes. Cussack et al. [40] have proposed an interpretation in LOTOS of the concept of inheritance. This relationship is not preserved by syntactic composition operators of

LOTOS so, they conclude, the language does not support object orientation. Some extensions of LOTOS to support inheritance have been proposed [130, 163], but their use is not widespread and it is not clear how they can benefit the specification of user interfaces. Inheritance is more useful as a feature of the implementation environment rather than for designing a user interface system. While it falls short of object-orientation, the ADC interactor model supports modularity and encourages reuse, as was pointed out in chapter 5. A more practical and better justified extension of the model that should facilitate its use would be a scheme for reusing ADC components, e.g. the polymorphic description of temporal constraints or a library of abstract data types. Their development is the subject of future work.

Validation of formal system models

Chapter 5 discussed the limitations of the case study as a scientific assessment of the model. It was argued that a thorough and objective evaluation of the ADC model shall be feasible at a further stage of its development. However, this assessment on its own is not the most important objective. Rather, more pressing questions concern the kinds of notations, models and tools which are needed and the direction of future research. It is more fruitful to test some of the assumptions and trade-offs incorporated in the model. For example, are LTSs a sufficiently expressive model or should a more expressive formal framework be adopted, e.g. to introduce true concurrency or time? Questions about the validity of the ADC model arise as well, concerning the relevance of formally specified or verified user interface properties to user interface development. Are decisions made throughout the design process adequately modelled using ADC interactors? Does the existence of an architectural specification facilitate the development of the user interface? These questions can be answered by studying the application of the model in forward engineering case studies, and by recording the tension between the design questions that need to be resolved and the capabilities of the modelling framework.

The question of the validity of the representations concerns also the analytical use of the model. Analytical results from other formal models of user interface software, both abstract and concrete, were reproduced in the framework of the ADC model. A criticism that applies to the thesis, as well as to the previous research results it integrates, is that they fail to establish the relevance of the properties modelled to the problem of user interface design. Many of the research approaches cited propose alternative formal models, discuss their features and their potential, and report exemplary applications in designing user interfaces. However, what seems to be missing from current research in the application of formal methods in human computer interaction is the empirical comparison of what can be expressed and verified using formal methods with what the designer needs to build into a specification, or between the former and the observations that emerge during the evaluation of the actual user interface. Such an investigation aided by the use of semi formal representations of the ADC interactor model is planned future work.

Tool support and visual representation

The question of using the model effectively prompts further research questions. Proposals are needed for making the best use of current tool support and for establishing the role of the formal specification in the development process. The earlier discussion on ADC as a conceptual framework suggests the need to develop informal notations to 'interface' to the formal specification. It is planned to investigate the feasibility of this concept. Alternatively, the formal specification may be facilitated by a visual specification language. In this thesis a set of conventions have been developed and used for the illustration of the ADC model. One avenue that will be investigated is to develop a visual (formal) specification language out of these conventions and editing tools to support this notation.

Future work could extend the process algebraic definitions of the transformations. Currently the decomposition transformation applies to a very restricted set of cases, and algorithms need to be developed to support the decomposition of interactors that share some state or to produce expressions involving dynamic composition operators. The development of software to support the transformations already described is planned in association with the editing facilities mentioned previously. Tool support could be developed to support verification, by helping to specify predicates for the verification of the user interface specification. These tools could provide an interface to the target formalism, e.g. ACTL or LOTOS.

Link to display models

Contrary to the concept of an ADC interactor described in chapter 4, the formal model of chapter 6 which supports the synthesis transformation, allows for interactors with multiple gates for display output. For example, if the list interactor and the scroll bar interactor of section 4.8 are synthesised, it would be desirable that a single output gate on the display side should offer the value of the combined display states for the two interactors. This synthesis of the display states concerns the specification of the data types rather than the process algebraic definition of synthesis. The ADC model abstracts away from any particular representation of the display, so it is applicable to any level of abstraction desired. However, by adopting a standard model for the display and the operations upon it, it would be possible to associate synthesis with the composition of display states. A single interactor modelling the screen could receive the display from a group of interactors and could compose their display values. Possible models of the display were mentioned in chapter 4, e.g. as a mapping of pixels to values [175], or as a set of regions [167]. An interesting problem for the future theoretical development of the formal interactor model is to experiment with such display models, aiming at an abstract representation that does not obscure the behavioural specification and which formalises properties of the display.

Developments in tool support and transformations for LOTOS

The previous point is closely related to another more practical observation. One of the arguments made in chapter 3 was that interface software is best specified using a hybrid notation. Accordingly, the ADC interactor model was shaped to help writing and manipulating full LOTOS specifications. Much of the complexity of the transformations stems from the interplay between the data component of the specification and the process algebraic component. This can be contrasted with chapter 7 where it can be observed that most uses of the model are easier when only basic LOTOS is discussed. So while (full) LOTOS describes more concisely the concept of an interactor it is hard to use in a practical context. The development of a model of the display may be some improvement for the model, but how practical it may be depends also on the capabilities of the general purpose tools for LOTOS.

Interactors and task based design

In the final part of chapter 7, the ADC interactor model was related to a model of the user's task knowledge. Task conformance was specified as a property of the temporal ordering of actions, as specified in the interface and the task model. Task conformance concerns also the task objects and their mapping to interface objects, the complexity of the task, etc. Research into a set of principles for relating task knowledge, or other user characteristics, to a user interface design would provide a solid theoretical foundation for the practical approach outlined in chapter 7. Presently, it seems a promising research project to consolidate and to develop further the proposed framework for testing and to examine its potential integration in the realistic development of user interfaces.

8.3 Contributions

The foremost contribution of the thesis is the development of the ADC interactor model. The thesis has focused primarily on the formal interactor model but has consistently developed and alluded to the concepts behind it. The formal representation and the conceptual framework it describes have been developed in parallel, with successive improvements and repeated attempts to describe these clearly.

ADC interactors have been defined as abstractions of software components that support the communication between the functional core of an interactive system and its user. The user interface system as a whole can be modelled as a single (monolithic) ADC interactor that communicates directly with the functional core and the user. Alternatively, the model may apply to a very small portion of the user interface software. In the latter case, the interface can be thought of as a composition of many ADC interactors. It is beneficial to switch between these two views during the design of a user interface, depending on what the focus of the design activity is: the structure of the software or its external behaviour. The ADC model describes a conceptual framework that can be used at all the intermediate levels of abstraction between an abstract external

view of an interface and a detailed construction oriented view. Its formal representation supports the transition from one view to the other, whether that be in a top down fashion or a bottom up fashion.

An important consideration of the thesis has been to contribute to addressing the problem of writing, reading and managing specifications of user interface software. The systematic reuse of formal abstractions and the use of existing and general purpose tool support have been advocated throughout. The ADC formal interactor model is one of a few formal interactor models that follow this policy, e.g. [150] also uses LOTOS and its tool support, and [145] uses Petri-Nets and general purpose tools for their verification. Other interactor models such as [55, 59], can be seen more as experimentation with alternative formal frameworks which emphasise the analytical use of their formalism independently of tool support. Interactors are specified by instantiating the ADC interactor model and the individually required adaptations are restricted to localised parts of the interactor specification. For example, the mapping between the dynamic component of the specification and the data specification component is fixed. While this does not reduce the expressiveness of the specification language, it results in a consistent specification style which, it has been argued, is easier to use and to understand than unstructured LOTOS.

Parameterisation was explored as a means of reusing common temporal behaviour specifications, i.e. starting, stopping, suspending, resuming and aborting the interactor. Other behaviours commonly encountered, e.g. continuous feedback, toggles, triggering behaviours, etc., were identified during the case study. The ADC model helps identify and organise such behaviours and further applications of the model should help compile a richer taxonomy of their specifications. These specifications can be easily reused by their composition (as opposed to parameterisation/actualisation) in the constraints component of an ADC interactor. In this case, LOTOS facilitates the specification because it supports multi-way synchronisation, which allows for the constraint oriented style of specification. In other cases, the synchronous communication of LOTOS was found restrictive, so ‘auxiliary’ components were defined to help specify the interface as a composition of smaller scale components (section 5.8). It was argued that the specifications of all the behaviours mentioned would benefit from tool and language support for their systematic reuse.

The main merit of the formal ADC interactor model is that it embodies the property of compositionality. To support this property a set of transformations have been defined (chapter 6). The synthesis transformation helps shape complex behaviour expressions involving ADC interactors into a single interactor. The decomposition transforms a monolithic ADC interactor into a behaviour expression involving simpler interactors. This ability to compose and decompose specifications maintaining the structure of the ADC interactor is a unique feature of this model that helps both the constructive and the analytical use of the specifications.

Another contribution of the thesis is the integration of diverse analytical results within the same conceptual and formal framework (chapter 7). The ADC interactor model

helps formalise predictability and observability related properties, which have been introduced with more abstract models of interactive systems. Section 7.1 includes an original formulation of predictability and observability properties which, it was argued, improves on earlier attempts. The ADC model also provides a framework for specifying and verifying dialogue properties, even when there is no separate dialogue component in the actual software specified, as is often the case (see the object-based architectures of chapter 2). The constructive specification of dialogue properties directly in LOTOS, and their verification by means of equivalence checking (section 7.2), is a further contribution of this thesis. The ADC interactor model combines the benefits of abstract models and dialogue specification notations. Other interactor models do not address the specification and verification of predictability and observability properties of the interface. Abstract models which are tailored for modelling this class of properties do not support the automatic verification of dialogue properties.

Finally, the thesis has described an approach to bridging the gap between software engineering system models and psychologically informed user models. As was mentioned in the introduction (chapter 1), this is a common objective for much research work in the field of human computer interaction. The framework of section 7.4 does not aspire to model or predict the phenomena surrounding interaction and does not make assumptions concerning user cognition and behaviour. It makes explicit and formalises intuitions which underlie task based design as mappings between formal representations of a user task model and an interface model, and helps define (partially) the requirement for task conformance. Task based design approaches try to establish this requirement by guiding the design process. A clear definition of this requirement is an important contribution of the thesis. The formal expression of task conformance is not particular to the ADC interactor model or to the TKS theory on which the formal user task model was based, although both models are well suited for describing the temporal dimension of a task and of the user interface behaviour. The definition of task conformance has prompted the proposal of a practical framework for testing user interface designs and their realisation with respect to the tasks they are intended to support. Potentially, this framework can bridge between formal models of user interfaces and practical techniques for user interface development.

In summary, the research reported has provided a deep and wide investigation into formal abstractions of user interface software. It has developed further the notion of a formal interactor model and in particular its use as an architectural abstraction. The ADC model improves on existing approaches, by integrating concepts from the domain of user interface software architectures, practical concerns about writing and using specifications, and ideas developed in the context of more abstract interaction models. The development of the ADC interaction model has opened up avenues for further research particularly aiming to enhance the applicability and relevance of formal methods to user interface design and development.

References

[1] Abowd GD (1992) Formal Aspects of Human Computer Interaction, PhD thesis, University of Oxford, Technical Report YCS 161, University of York.

[2] Abowd GD, Coutaz J & Nigay L (1992) Structuring the Space of Interactive system Properties, In Larson J & Unger C (Eds.) Engineering for Human Computer Interaction. Proceedings of the IFIP TC2/WG2.7 working conference, IFIP transactions A-18, Elsevier (North-Holland), pp. 113-129

[3] Abowd GD, Wang HM & Monk A (1995) A formal technique for automated dialogue development. In Olson GM & Schuon S (Eds.) DIS'95 Conference Proceedings. ACM Press, pp. 219-226.

[4] Abramsky S (1987) Observation Equivalence as a Testing Equivalence. Theoretical Computer Science, Vol. 53, No. 198, pp. 225-241.

[5] Accot J, Chatty S & Palanque P (1996) A formal description of low level interaction and its application to multimodal interactive systems. In Bodart F & Vanderdonckt J (Eds.) Design, Specification and Verification of Interactive Systems '96, Springer (Wien), pp. 92-104.

[6] ACM SIGCHI (1992) ACM Special Interest Group on Human-Computer Interaction. Curriculum Development Group. ACM SIGCHI curricula for human-computer interaction. Technical report, ACM, New York.

[7] Addison M & Thimbleby H (1994) Manuals as structured programs. In Cockton G, Draper SW & Wier GRS (Eds.) People and Computers IX, Proceedings of HCI'94, Glasgow, 1994, Cambridge University Press, pp. 67-80.

[8] Alexander H (1990) Structuring dialogues using CSP. In Harrison MD & Thimbleby HW (Eds.) Formal Methods in Human Computer Interaction, Cambridge University Press, pp. 273-295.

[9] Apple Computer Inc. (1993) Inside Macintosh. QuickTime™. Addison Wesley.

[10] Baeten JCM & Weijland WP (1990) Process Algebra. Cambridge University Press.

[11] Barnard PJ (1987) Cognitive Resources and the learning of Human-Computer Interaction. In Caroll MJ (Ed.) Interfacing Thought-Cognitive aspects of Human-Computer Interaction. MIT Press, pp. 112-158.

[12] Barnard PJ & Harrison MD (1992) Towards a Framework for Modelling Human-Computer Interactions. In Gornostaev J (Ed.) Proceedings of the East-West International Conference on Human-Computer Interaction EWHCI'92, St. Petersburg, Russia, 4-8 August, 1992, International Centre for Scientific and Technological Information, Moscow, pp. 189-197.

[13] Barnard PJ & May J (1994) Interactions with advanced graphical interfaces and the deployment of latent human knowledge. In Paternó F (Ed.) Interactive Systems: Design, Specification and Verification. Springer (Wien), pp. 15-49.

[14] Bass L, Coutaz J & Unger C (1992) A reference model for interactive system construction. In Gornostaev J (Ed.) Proceedings of the East-West International Conference on Human-Computer Interaction EWHCI'92, St. Petersburg, Russia, 4-8 August, 1992, International Centre for Scientific and Technological Information, Moscow, pp 23-30.

[15] Bass L & Coutaz J (1991) Developing Software for the User Interface, Addison-Wesley, USA.

[16] Bernadeschi C, Fantechi A & Paternó F (1995) Application of correctness preserving transformations for deriving architectural descriptions of interactive systems from user interface specifications. In Proceedings SEKE'95, The 7th International Conference on Software Engineering and Knowledge Engineering, June 22-24, 1995, Knowledge Systems Institute, Illinois.

[17] Biljon VWR (1988) Extending Petri Nets for specifying man-machine dialogues. International Journal of Man Machine Studies, Vol. 28, pp. 437-455.

[18] Bolognesi T & Brinksma E (1989) Introduction to the ISO specification language LOTOS. Van Eijk P, Vissers C & Diaz M (Eds.) The Formal Description Technique LOTOS, Elsevier (North-Holland), pp. 23-73.

[19] Bolognesi T, De Frutos-Escrig D & Ortega-Mallen Y (1991) Graphical Composition Theorems for Parallel and Hiding Operators. In Quemada J, Mañas J & Vazquez E (Eds.) Formal Description Techniques III, Elsevier (North-Holland), pp. 459-470.

[20] Bolognesi T, De Frutos D, Langerak R & Latella D (1995) Correctness preserving transformations for the early phases of software development. In Bolognesi T, van de Lagemaat J & Vissers C (Eds.) LOTOSphere: Software Development with LOTOS. Kluwer (Netherlands), pp. 161-180.

[21] Bornat R & Thimbleby H (1989) The life and times of ded. In Long J & Whitefield A (Eds.) Cognitive Ergonomics and Human-Computer Interaction. Cambridge University Press, pp. 225-255.

[22] Bouali A, Gnesi S & Larosa S (1994) The Integration Project for the JACK Environment. Bulleting of the EATCS, No 54, pp 307-223..

[23] Bouajjani A, Fernadez JC, Graf S, Rodríguez C & Sifakis J (1991) Safety for branching time semantics. In Leach Albert J, Monien B, Rodríguez Artalejo (Eds.) Automata, Languages and Programming, 18th International Colloquium, Madrid Spain, July 1991, LNCS 510, Springer-Verlag, pp. 76-92.

[24] Bowen JP & Hinchey MG (1995) Seven more myths of formal methods. IEEE Software, Vol. 12, No. 4, pp. 34-41.

[25] Bowen JP & Hinchey MG (1995) Ten commandments of formal methods. IEEE Computer, Vol. 28, No. 4, pp. 56-63.

[26] Bowen JP & Stavridou V (1993) The industrial take-up of formal methods in safety-critical and other areas: a perspective. In Woodcock JCP & Larsen PG (Eds.) Formal Methods Europe 1993: Industrial Strength Formal Methods, Springer-Verlag, LNCS 670, pp. 183-195.

[27] Brinksma E (1989) A theory for the derivation of tests. In van Eijk PHJ, Vissers CA & Diaz M (Eds.) The Formal Description Technique Lotos, Elsevier (North-Holland), pp. 235-247.

[28] Brinksma E & Langerak R (1995) Functionality decomposition by compositional correctness preserving transformation. South African Computer Journal, SAJC/SART, No. 13, pp. 2-13.

[29] Brinksma E, Scollo G & Steenbergen C (1987) LOTOS specifications, their implementations and their tests. In Sarikaya B & Bochman (Eds.) Protocol Specification, Testing and Verification VI, Elsevier (North-Holland) IFIP, pp. 349-360.

[30] Caneve M & Salvatori E (1992) LITE user manual. LOTOSPHERE Project Technical Report, Lo/WP2/N0034/Vo8.

[31] Card SK, Mackinlay JD & Robertson GG (1991) A morphological analysis of the design space of input devices. ACM Transactions on Information Systems, Vol. 9, No. 2, pp. 99-122.

[32] Card SK, Moran TP & Newell A (1983) The Psychology of Human-Computer Interaction, Lawrence Erlbaum Associates (Hillsdale NJ).

[33] Cardelli L & Pike R (1985) Squeak: a Language for Communicating with Mice. In SIGRAPH ’85 conference proceedings, ACM Computer Graphics, Vol. 19, No. 3, 199-204.

[34] Carrol JM (1990) Infinite detail and emulation in an ontologically minimized HCI. In Chew JC & Whiteside J (Eds.) Empowering People-CHI'90 conference Proceedings. ACM Press, pp. 321-327.

[35] Chi UH (1985) Formal Specifications of User Interfaces: A Comparison and Evaluation of Four Axiomatic Approaches. IEEE Transactions on Software Engineering, Vol. 11, No. 8, pp. 671-685.

[36] Clark RG (1992) LOTOS Design-Oriented Specifications in the Object Based Style, Technical Report 84, Dept. of Computer Science and Mathematics, University of Stirling.

[37] Cockton G (1990) The architectural bases of design re-use. In Duce D A, Gomes MR, Hopgood FRA & Lee JR (Eds.) User Interface Management and Design. Proceedings of the workshop on user interface management systems and environments. Lisbon, June 1990. Springer-Verlag, pp. 15-34.

[38] Cohen B, Harwood WT & Jackson M (1986) The Specification of Complex Systems. Addison-Wesley.

[39] Coutaz J (1987) PAC, an Object Oriented Model for Dialog Design. Bullinger HJ & Shakiel B (Eds.) INTERACT'87 Conference Proceedings, Elsevier (North-Holland), pp. 431-436.

[40] Cussack E, Rudkin S & Smith C (1990) An object oriented interpretation of LOTOS. In Vuong ST (Ed.) Formal Description Techniques II, Elsevier (North Holland), pp. 211-226.

[41] Dance JR, Granor TE, Hill RD, Hudson SE, Meads J, Myers BA & Schulert A, The Run time structure of UIMS Supported Applications, Computer Graphics, Vol. 21, No. 2, 1987, pp. 97-101.

[42] De Bruin H, Bouwman P & van den Bos J (1994) Modelling and analysing human-computer dialogues with protocols. In Paternó F (Ed.) Interactive Systems: Design Specification and Verification, Springer (Wien), pp. 95-116.

[43] De Bruin H (1995) DIGIS A Model-Based Graphical User Interface Design Environment for Non-Programmers. PhD thesis, Erasmus Universiteit Rotterdam.

[44] De Meer J, Roth R & Vuong S (1992) Introduction to algebraic specifications based on the language ACT ONE. Computer Networks and ISDN Systems, Vol. 23, pp. 363-392

[45] De Nicola R (1989) Extensional Equivalences for Transition Systems, Acta Informatica Vol. 24, 211-237.

[46] De Nicola, Fantechi A, Gnesi S & Ristori (1993) An action-based framework for verifying logical and behavioural properties of concurrent systems. Computer Networks and ISDN Systems, Vol. 25, pp. 761-778.

[47] De Nicola R & Hennessy MCB (1984) Testing Equivalence for Processes. Theoretical Computer Science, North Holland, Vol. 34, pp. 83-133.

[48] De Nicola R & Vaamdrager F (1990) Action versus State based Logics for Transition Systems. In Guessarian I (Ed.) Semantics of Systems of Concurrent Processes, Springer, Verlag, LNCS 469, pp 407-419.

[49] Dix AJ (1991) Formal Methods for Interactive Systems, Academic Press.

[50] Dowell J & Long J (1989) Towards a conception for an engineering discipline of human factors. Ergonomics, Vol. 32, No. 11, pp. 1513-1535.

[51] Duce DA (1995) Users: Summary of working group discussion. In Paternó F (Ed.) Interactive Systems: Design Specification and Verification, Springer 1995, pp. 51-56.

[52] Duce DA, ten Hagen PJW & van Liere R (1989) Components, Frameworks and GKS Input. In Hansmann W, Hopgood F & Strasser W (Eds.) Eurographics'89 Conference Proceedings, Elsevier (North-Holland), pp. 87-103.

[53] Duce DA, ten Hagen PJW & van Liere R (1990) An approach to hierarchical input devices. Computer Graphics Forum, Vol. 9, No. 1, pp. 15-26.

[54] Duke D, Faconti F, Harrison MD & Paternó F (1993) Unifying Views of Interactors. In Proceedings of the Workshop on Advanced Visual Interfaces '94, Bari, June1994, ACM Press, pp. 143-152.

[55] Duke DJ & Harrison MD (1993) Abstract Interaction Objects. In Hubbold RJ., Juan R (Eds.) EUROGRAPHICS'93, Computer Graphics Forum, Vol. 12, No.3, pp. 26-36.

[56] Duke DJ & Harrison MD (1994) A theory of presentations. In Naftalin M, Denvir T & Bertran M (Eds.) Proceedings Formal Methods Europe '94, Industrial Benefit of Formal Methods, Springer-Verlag, LNCS 873, pp. 271-290.

[57] Duke DJ & Harrison MD (1994) From Formal Models to Formal Methods. In Taylor RN & Coutaz J (Eds.) Software Engineering and Human-Computer Interaction. ICSE'94 Workshop on Software Engineering and Human Computer Interaction, Springer-Verlag, LNCS 896, pp. 159-173.

[58] Duke DJ & Harrison MD (1994) Folding Human Factors into Rigorous Development. In Paternó F (Ed.) Interactive Systems: Design, Specification and Verification. Springer, pp. 333-350.

[59] Duke DJ & Harrison MD (1995) Event model of human-system interaction. Software Engineering Journal. Vol. 10, No. 1, pp. 3-12.

[60] Duke DJ & Harrison MD (1995) Interaction and task requirements. In Palanque P, Bastide R (Eds.) Design, Specification, Verification of Interactive Systems '95, Springer Wien, pp. 54-75.

[61] Ehrig H & Mahr B (1985) Fundamentals of Algebraic Specification 1, Springer Verlag.

[62] Faconti GP (1993) Towards the Concept of Interactor. AMODEUS project report, ref. sm/wp8.

[63] Faconti GP & Duke DJ (1996) Device Models. In Bodart F & Vanderdonckt J (Eds.) Design, Specification and Verification of Interactive Systems '96, Springer (Wien), pp.73 -91.

[64] Faconti GP, Fornari A & Zani N (1994) Visual Representation of Formal Specification: An Application to Hierarchical Input Devices. In Paternó F (Ed.) Interactive Systems: Design, Specification and Verification. Springer, pp. 349-368.

[65] Faconti GP & Paternó F (1990) An approach to the formal specification of the components of an interaction. In Vandoni CE & Duce DA (Eds.) Eurographics'90 Conference Proceedings. Elsevier (North-Holland) pp. 481-494.

[66] Faconti GP & Paternó F (1992) The input model of standard graphics systems revisited by formal specification. In Kilgour A & Kjelldahl L (Eds.) Eurographics'92 Conference, Computer Graphics Forum, Vol. 11, No. 3, pp. 237-251.

[67] Fantechi A, Gnesi S, Mazzarini G (1991) How expressive are LOTOS behaviour expressions? In Quemada J, Mañas J, Vásquez E (Eds.) Formal Description Techiques III, Elsevier (North-Holland), IFIP, pp. 17-32.

[68] Fernadez JC, Caravel H, Mounier L, Rasse A, Rodríguez C & Sifakis J (1992) A toolbox for the verification of LOTOS Programs. 14th International Conference on Software Engineering, Melbourne, May 1992.

[69] Fernadez JC, Jard C, Jéron T & Viho C (1996) Using on-the-fly verification techniques for the generation of test suites. In Alur R & Henzinger TA (eds.) 8th International Conference on Computer Aided Verification: (CAV'96), Springer-Verlag (Berlin), LNCS 1102, pp. 348-359.

[70] Ferro G (1994) Un model checker lineare per la logica temporale ACTL. Tesi di Laurea, Università degli studi di Pisa.

[71] Fields RE, Wright PC & Harrison MD (1995) A Task Centred Approach to Analysing Human Error Tolerance Requirements. In Harrison MD & Zave P (Eds.) Requirements Engineering '95. IEEE Computer Society Press, pp. 18-26.

[72] Foley JD, Wallace VL & Chan P (1984) The human factors of Computer Graphics Interaction Techniques, IEEE Computer Graphics and Applications, Vol. 4, No. 11, pp13-48.

[73] Gaudel MC (1994) Formal Specification Techniques. 16th International Conference on Software Engineering (ICSE'94), Sorrento, Italy, pp. 223-227.

[74] Glass RL (1996) Formal methods are a surrogate for a more serious software concern. In IEEE Computer, Vol. 29, No. 4, pp. 19.

[75] Goldberg A & Robson D (1983) Smalltalk-80. The Language and its Implementation. Addison-Wesley.

[76] Gotzhein R (1987) Specifying Abstract Data Types with Lotos. In Sankaya B & Bochman EV (Eds.) Protocol Specification Testing and Verification VI, Elsevier (North-Holland), pp. 15-26.

[77] Green M (1985) Report on Dialogue Specification Tools. In Pfaff GE, User Interface Management Systems, Springer-Verlag, pp. 9-20.

[78] Green M (1986) A Survey of Three Dialogue Models. ACM Transactions on Graphics, Vol. 5, No. 3, pp. 244-275.

[79] de Haan G (1994) An ETAG based approach to the design of user interfaces. Proceedings of the 15th Interdisciplinary Workshop on Informatics and Pscychology. Scharding 1994.

[80] Hall JA (1990) Seven Myths of Formal Methods. IEEE Software, Vol. 7 No. 5, pp. 11-19.

[81] Harel D (1987) Statecharts: A Visual Formalism for complex systems. Science of Computer Programming Vol. 8, No. 3, pp. 231-274.

[82] Harrison MA (1978) Introduction to Formal Language Theory. Addison Wesley.

[83] Harrison MD (1992) A model for the option space of interactive systems. In Larson J & Unger C (Eds.) Engineering for Human Computer Interaction. Proceedings of the IFIP TC2/WG2.7 working conference, IFIP transactions A-18, Elsevier (North-Holland), pp. 155-170.

[84] Harrison MD & Dix AJ (1990) A state model of direct manipulation in interactive systems. In Harrison MD & Thimbleby HW (Eds.) Formal Methods in Human Computer Interaction, Cambridge University Press, pp. 129-151.

[85] Harrison MD & Thimbleby HW (1985) Formalising Guidelines for the Design of Interactive Systems. In Johnson P & Cook S (Eds.) People and Computers: Designing the Interface, Proceedings BCS-HCI'85 Conference, Cambridge University Press, pp. 161-171.

[86] Harrison MD & Duke DJ (1994) A review of formalisms for describing interactive behaviour. Taylor RN & Coutaz J (Eds.) Software Engineering and Human-Computer Interaction. ICSE'94 Workshop on Software Engineering and Human Computer Interaction, Springer-Verlag, LNCS 896, pp. 49-75.

[87] Hartson RH & Boehm-Davis D (1993) User interface development processes and methodologies. Behaviour and Information Technology, Vo. 12, No. 2, pp. 98-114.

[88] Hartson RH & Hix D (1989) Human-computer interface development: concepts and systems for its management, ACM Computing Surveys, Vol. 21, No. 1, pp. 5-92.

[89] Hartson RH & Mayo KA (1994) A framework for precise, reusable abstractions. In Paternó F (Ed.) Interactive systems: design, specification and verification, Springer, pp. 49-484.

[90] Hartson RH, Siochi AC & Hix D (1990) The UAN: A user oriented representation for direct manipulation systems. ACM Transactions on Information Systems, Vol. 8, pp. 181-203.

[91] Hekmapoutr S & Ince D (1987) Evolutionary prototyping and the human computer interface. In Bullinger HJ & Shakiel B (Eds.) INTERACT '87 conference proceedings, Elsevier(North-Holland), pp. 479-484.

[92] Hill RD (1986) Supporting concurrency, communication, and synchronization in human computer interaction-the Sassafras UIMS, ACM Transactions on Graphics, Vol. 5, No. 3, pp. 179-210.

[93] Hill RD & Herrmann M (1987) The Structure of Tube-A tool for implementing advanced user interfaces. In Proceedings Eurographics '89, Elsevier (North-Holland), pp. 12-25.

[94] Hill RD & Hermann M (1990) The composite object user interface architecture. In Duce DA, Gomes MR, Hopgood FRA & Lee JR (Eds.) User Interface Management and Design. Proceedings of the workshop on user interface management systems and environments. Lisbon, June 1990. Springer-Verlag, pp. 257-271.

[95] Hill RD, Brinck T, Rohall SL, Patterson JF & Wilner W (1994) The Rendezvous architecture and language for constructing multiuser applications, ACM Transactions on Computer Human Interaction, Vol. 1, No. 2, pp. 81-125.

[96] Hoare CAR (1985) Communicating Sequential Processes. Prentice Hall International.

[97] Holloway CM & Butler R (1996) Impediments to the industrial use of formal methods. IEEE Computer, Vol. 29, No. 4, pp. 26-27.

[98] Hussey A & Carrington D (1996) Using Object-Z to compare the MVC and PAC architectures. In Roast C & Siddiqi J (Eds.) Formal Aspects of the Human Computer Interface, BCS-FACS workshop , Springer eWiC series.

[99] ISO (1985) Information processing systems-computer graphics-graphical kernel system (GKS) ISO 7942, ISO General Secretariat.

[100] ISO (1989) Information Processing Systems-Open Systems Interconnection-LOTOS-A Formal Description Technique based on the Temporal Ordering of

Observational Behaviour, ISO/IEC 8807, International Organisation for Standardisation, Geneva.

[101] ISO (1997) Enhancements to LOTOS, ISO/IEC, Ref. JTC1/SC21/WG7. Project WI 1.21.20.2.3.

[102] Jacob RKJ, Leggett JL, Myers BA & Pausch R (1993) Interaction styles and input/output devices. Behaviour and information technology, Vol. 12, No. 2, pp. 69-79.

[103] Jacob RKJ (1986) A Specification Language for Direct-Manipulation User Interfaces. ACM Transactions on Graphics, Vol. 5, No. 4, pp. 283-317.

[104] Johnson CW (1995) The application of Petri-Nets to reason about human factors problems during accident analysis. In Palanque P, Bastide R (Eds.) Design, Specification, Verification of Interactive Systems '95, Springer Wien, pp. 93-112.

[105] Johnson CW & Harrison MD (1990) PRELOG-A system for presenting and rendering logic specifications of interactive systems. In Vandonis CE & Duce D (Eds.) Eurographics '90 Conference Proceedings, Elsevier (North Holland), pp. 469-480.

[106] Johnson CW & Harrison MD (1992) Using temporal logic to support the specification and prototyping of interactive control systems. Interactional Journal of Man-Machine Studies, Vol. 37, pp. 357-385.

[107] Johnson CW (1996) The evaluation of user interface notations. In Bodart F & Vanderdonckt J (Eds.) Design, Specification and Verification of Interactive Systems '96, Springer (Wien), pp. 188-206..

[108] Johnson P (1992) Human Computer Interaction. Psychology-Task Analysis and Software Engineering, McGraw-Hill (London).

[109] Johnson H & Johnson P (1991) Task knowledge structures: Psychological basis and integration into systems design. Acta Psychologica, Vol. 78, pp. 3-26.

[110] Johnson P, Johnson H, Waddington R. & Shouls A (1988) Task Related Knowledge Structures: Analysis, Modelling and Application, People and Computers IV , Cambridge University Press, pp. 35-61.

[111] Johnson P, Wilson S, Markopoulos P & Pycock J (1993) ADEPT-Advanced Design Environment for Prototyping with task models, Demonstration abstract. In Aschlund S, Mullet K, Henderson A, Hollnagel E & White T (Eds.) Bridges Between Worlds-INTERCHI '93 conference proceedings, Addison-Wesley, pp. 56.

[112] Kovacevic S (1992) A compositional model of human-computer dialogues. In Blattner M. & Dannenberg R (Eds.) Multimedia Interface Design, ACM Press, Addison-Wesley, pp. 373-404.

[113] Krasner GE & Pope ST (1988) A Cookbook For Using the Model-View-Controller User Interface Paradigm in The Smalltalk-80 System, Journal of Object Oriented Programming, Vol. 1, No. 3, pp. 26-49.

[114] Lim KY & Long J (1994) The MUSE method for usability engineering. Cambridge University Press, Glasgow.

[115] Liskov B & Guttag J (1986) Abstraction and Specification in Program Development. MIT-Press, Cambridge-Massachusetts.

[116] Logrippo L, Faci M & Haj-Hussein M (1992) An Introduction to LOTOS: learning by examples, Computer Networks and ISDN Systems, Vol. 23, No. 5, pp. 325-342.

[117] LOTOSPHERE (1990) A theoretical and methodological framework to conformance testing. Burmeister J & de Meer J (Eds.) project deliverable Lo/WP1.T1.3/N0006/V5, ESPRIT ref. 2304.

[118] LOTOSPHERE (1991) Catalogue of LOTOS Correctness Preserving Transformation. In Bolognesi T (Ed.) Final Deliverable Lo/WP1/T1.2 /N0045 /Vo3.

[119] Mañas JA (1995) Getting to use the LOTOSphere Integrated Tool Environement (LITE). In Bolognesi T, van de Lagemaat J & Vissers C (Eds.) LOTOSphere: Software Development with LOTOS. Kluwer (Netherlands), pp. 87-107.

[120] Markopoulos P (1992) The Adept models and formal description techniques. Technical report from the ADEPT project, ref. Adept/D/3/1/2.4.92, DTI IED 4/1/1573.

[121] Markopoulos P, Wilson S, Pycock J & Johnson P (1991) Formal specifications and task based user interface design. In Paternó F (Ed.) Formal Methods in Computer Graphics, Eurographics Workshop, Marina di Carrara, 17-19 June, 1991.

[122] Markopoulos P & Gikas S (1994) Towards A Formal Model for Extant Task Knowledge Representation. In Stary C (Ed.) 1st Interdisciplinary Workshop on Cognitive Modelling and User Interface Development, Vienna, December 15-17, 1994.

[123] Markopoulos P (1995) On the Expression of Interaction Properties within an Interactor Model. In Palanque P & Bastide R (Eds.) Design, Specification, Verification of Interactive Systems ’95, Springer (Wien), pp. 294-311.

[124] Markopoulos P (1996) Case study in the formal specification of the SimplePlayer™ graphical interface for playing QuickTime™ movies, using the ADC interactor model. Technical Report 712, Dept. of Computer Science, QMW College, University of London.

[125] Markopoulos P, Rowson J & Johnson P (1996) On the composition of interactor specifications. In Roast C & Siddiqi J (Eds.) Formal Aspects of the Human Computer Interface, BCS-FACS workshop, Springer, eWiC series.

[126] Markopoulos P, Rowson J & Johnson P (1996) Dialogue design in the framework of an interactor model. In Bodart F & Vanderdonckt J (Eds.) Design, Specification and Verification of Interactive Systems '96, workshop informal proceedings, pp. 231-244.

[127] Markopoulos P, Wilson S, Pycock J & Johnson P (1992) Adept-A task based design environment. In Shriver BD (Ed.) 25th Hawaii International Conference on System Sciences, Conference Proceedings, Vol. II, IEEE Computer Society Press (California), pp. 587-596.

[128] Markopoulos P, Wilson S & Johnson P (1994) Representation and Use of Task Knowledge in a User Interface Design Environment. IEE Proceedings~E, Computers and Digital Techniques, Vol. 141, No. 2, pp. 79-84.

[129] Marshall LS (1986) A formal Description for User Interfaces, PhD thesis, University of Manchester.

[130] Mayr T (1989) Specification of Object-Oriented Systems in LOTOS. In Turner KJ (Ed.) Formal Description Techniques, Elsevier(North-Holland), pp. 107-119.

[131] Meyer B (1988) Object-oriented software construction. Prentice-Hall International (UK).

[132] Mezzanotte M & Paternó (1995) Including Time in the Notion of Interactor. In Johnson C & Gray P. (Eds.) Proceedings from a Workshop on temporal Aspects of Usability. CIST Technical Report, Glasgow University.

[133] Milner R (1989) Communication and Concurrency. Prentice Hall, UK.

[134] Monk A, Wright P, Haber J & Davenport (1993) Improving your human computer interface: a practical technique. Prentice-Hall (Hemel-Hempstead).

[135] Moran TP (1981) The command language grammar: a representation for the user interface of interactive computer systems, International Journal of Man-Machine Studies, Vol. 15, pp. 3-50.

[136] Myers BA (1990) A new model for handling input. ACM Transactions on Information Systems. Vol. 8, No. 3, pp. 289-320.

[137] Myers BA, Giuse DA, Dannenberg RB, Zanden BV, Kosbie DS, Pervin E, Mickish A& Marchal P (1990) Garnet-Comprehensive support for graphical highly interactive user interfaces. IEEE Computer, Vol. 23, No. 11, pp. 71-85.

[138] Myers BA (1993) Why are Human-Computer-Interfaces Difficult to Design and Implement? Technical report, Carnegie Mellon University, CMU-CS-93-183.

[139] Myers BA & Rosson MB (1992) Survey on user interface programming. In Bauersfeld P, Bennett J & Lynch G (Eds.) Striking a Balance, CHI'92 Conference Proceedings, ACM Press, pp. 195-202.

[140] Najm E & Stefani JB (1992) Dynamic Configuration in LOTOS. In Parker KR & Rose GA (Eds.) Formal description techniques IV, Elsevier (North-Holland), pp. 201-216.

[141] Nielsen J (1995) Scenarios in discount usability engineering. In Carrol JM (Ed.) Scenario-based design: envisioning work and technology in system development, Wiley, pp. 59-83.

[142] Olsen D (1990) Propositional production systems for dialogue description. In Chew JC & Whiteside J (Eds.) Empowering people - CHI'90 conference proceedings, Human Factors in Computing Systems, ACM (New York), pp. 57-63.

[143] Olsen D, Monk A & Curry M (1995) Automatic Dialogue Analysis. Human Computer Interaction, Vol. 10, No. 1, pp. 40-78.

[144] Palanque P, Bastide R, Dourte L & Sibertin B (1993) Design of User Driven Interfaces Using Petri Nets and Objects. In Rolland C, Bodart F & Couvert C (Eds.) Advanced Information Systems Engineering, Paris, Springer-Verlag, LNCS 685, Springer, pp. 569-585.

[145] Palanque P & Bastide R (1995). Petri net based design of user-driven interfaces using the interactive cooperative objects formalism. In Paternó F (Ed.) Interactive Systems: Design Specification and Verification, Springer 1995, pp. 383-400.

[146] Palanque P & Bastide R (1996) A design life-cylce for the formal design of user interface. In Roast C & Siddiqi J (Eds.) Formal Aspects of the Human Computer Interface, BCS-FACS workshop , Springer, eWiC series

[147] Palanque P, Paternó F, Bastide R & Mezzanotte M (1996) Towards an integrated proposal for Interactive Systems design based in TLIM and ICO. In Bodart F & Vanderdonckt J (Eds.) Design, Specification and Verification of Interactive Systems '96, Springer (Wien), pp. 162-187.

[148] Paternó F (1993) A formal approach to the evaluation of interactive systems. SIGCHI Bulletin, Vol. 26, No. 2, pp. 69 -73.

[149] Paternó F (1994) A Theory of User Interaction Objects. Journal of Visual Languages and Computing, Vol. 5, pp. 227-249.

[150] Paternó F & Faconti G (1992) On the use of LOTOS to describe graphical interaction. In Monk A, Diaper D & Harrison MD, People and Computers VII, Proc. HCI'92 Conference, Cambridge University Press, pp. 155-173.

[151] Paternó F (1993) Definition of properties of user interfaces using action based temporal logic. In Proceedings, 5th International Conference in Software Engineering and Knowledge Engineering, San Francisco, June, pp. 314-319.

[152] Paternó F & Mezzanotte M (1995) Analysing Matis by Interactors and ACTL. Amodeus Project Document: System Modelling/WP36.

[153] Paternó F & Mezzanotte M (1995) Formal analysis of user and system interactions in the CERD. Case Study. Amodeus project document: System modelling System Modelling/WP48.

[154] Paternó F & Leonardi A (1994) A Semantics-based Approach for the Design and Implementation of Interaction Objects, (Eurographics'94), Computer Graphics Forum, Blackwell , Vol. 13, No. 3, pp. 35-53.

[155] Paternó F, Sciacchitano MS & Lowgren J (1995) User Interface Evaluation Mapping Physical User Actions to Task-Driven Formal Specifications. In Palanque P & Bastide R (Eds.) Design, Specification, Verification of Interactive Systems '95, Springer Wien, pp. 294-311.

[156] Pavón S & Larrabeiti D (1993) LOLA (LOtos LAboratory) User Manual v.3.4, http://www.dcs.upm.es/~lotos.

[157] Quemada J, Ferreira Pires J, Mañas JA, Azcorra A & Robles T (1993) Introduction to LOTOS. In Turner K (Ed.) Using formal description techniques - an introduction to Estelle, LOTOS and SDL, Wiley, pp. 47-83.

[158] Quemada J, Azcorra A & Pavón (1993) Software Development with LOTOS. In Turner K (Ed.) Using formal description techniques - an introduction to Estelle, LOTOS and SDL, Wiley, pp. 345-373.

[159] Roast CR (1993) Executing Models in Human Computer Interaction. PhD Thesis. Department of Computer Science. The University of York.

[160] Roast C R & Harrison M W & Wright P (1989) Complementary methods for the iterative design of interactive systems. In Salvendy G & Smith M (Eds.) Designing and Using Human Computer Interfaces and Knowledge Based Systems, Elsevier , pp. 651-658.

[161] Roast, C. (1994) Modelling Interaction Using Template Abstractions. In Cockton G, Draper SW & Weir GRS (Eds.) People and Computers IX, Cambridge University Press, pp. 273-285.

[162] Rosson MB, Maass S & Kellogg WA (1988) The designer as user: building requirements for design tools from design practice. Communications of the ACM, Vol. 31, No. 11, pp. 1288-1298.

[163] Rudkin S (1992) Inheritance in LOTOS. In Parker KR & Rose GA (Eds.) Formal description techniques IV, Elsevier (North-Holland), pp. 409-424.

[164] Ruid-Delgado A, Pitt D & Smythe C (1995) A review of object oriented approaches in formal methods. The Computer Journal, Vol. 38, No. 10, pp. 777-784.

[165] Runciman C (1990) From abstract models to functional prototypes. Harrison MD & Thimbleby HW (Eds.) Formal Methods in Human Computer Interaction, Cambridge University Press, pp. 201-232.

[166] Saiedian H (Ed.) (1996) An invitation to formal methods. IEEE Computer, Vol. 29, No. 4, pp. 16-30. This is a 'roundtable' discussion, or rather a suite of papers from several authors.

[167] Samuel J (1996) Constraint programming for user interface construction. PhD thesis. Queen Mary and Westfield College, University of London.

[168] Sufrin B (1982) Formal Specification of a Display-Oriented Text Editor. Science of Computer Programming, North Holland, Vol. 1,pp. 157-202.

[169] Sufrin B & He J (1990) Specification analysis and refinement of interactive processes. Harrison MD & Thimbleby HW (Eds.) Formal Methods in Human Computer Interaction, Cambridge University Press, pp. 153-200.

[170] Thimbleby HW (1984) Generative user-engineering principles for user interface design. In Shackel B (Ed.) Proceedings INTERACT'84, North-Holland, pp. 661-666.

[171] Thimbleby HW (1994) Formulating Usability. SIGCHI Bulletin, Vol. 26, No. 2, pp. 59-64.

[172] Took R (1990) Putting design into practice: formal specification and the user interface. In Harrison MD & Thimbleby H (Eds.) Formal Methods in Human-Computer Interaction. Cambridge University Press, pp. 63-96.

[173] Took R (1990) Surface interaction: a paradigm and model for separating application and interface. In Chew JC & Whiteside J (Eds.) Human Factors in Computing Systems - CHI'90 conference proceedings, ACM Press (New-York), pp. 35-42.

[174] Took R (1994) Understanding direct manipulation algebraically. In Paternó F (Ed.) Interactive Systems: Design, Specification and Verification, Springer, pp. 413-428.

[175] Torres JC & Lares B (1995) Using an abstract model for the formal specification of interactive graphic systems. In Paternó F (Ed.) Interactive Systems: Design Specification and Verification, Springer, pp. 429-444.

[176] Turner K (1987) An Architectural Semantics for LOTOS. In Rudin H & West CH (Eds.) Protocol Specification Testing and Verification VII, Elsevier (North Holland), pp. 15-28.

[177] Turner K (Ed.) (1993) Using formal description techniques - an introduction to Estelle, LOTOS and SDL. Wiley.

[178] Van Eijk PHJ, Vissers CA & Diaz M (Eds.) The Formal Description Technique Lotos, Elsevier (North-Holland).

[179] Vissers CA, Scollo G & van Sinderen M (1988) Architecture and Specification Style in Formal Descriptions of Distributed Systems. In Aggarwal S & Sabnani K (Eds.) Protocol Specification Testing and Verification VIII, Elsevier (North-Holland), pp. 189-204.

[180] Vissers CA, Scollo G, van Sinderen M & Brinksma E (1991) Specification styles in distributed systems design and verification. Theoretical Computer Science, Vol. 89, pp. 179-206.

[181] The UIMS Tool Developers Workshop (1992) A Metamodel for the runtime architecture of an interactive system. SIGCHI Bulletin, Vol. 24, No. 1, pp. 32-37.

[182] Wasserman W (1985) Extending State Transition Diagrams for the Specification of Human Computer Interaction. IEEE Transactions on Software Engineering, Vol. 11, No. 8, pp. 699-713.

[183] Wellner PD (1989) Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. CHI'89 Conference Proceedings, ACM Press, pp. 177-182.

[184] Wezeman CD (1990) The CO-OP method for compositional derivation of canonical testers. In Brinksma E, Scollo G & Vissers CA (Eds.) Protocol Specification, Testing and Verification IX, Elsevier (North-Holland), pp. 145-158.

[185] Whitefield A (1994) Comparative analysis of tasks analysis products. Interacting with Computers, Vol. 6, No. 3, pp. 289-309.

[186] Whiteside J, Bennet J & Holtzblatt J (1988) Usability Engineering: Our experience and evolution. In Helander M (Ed.) Handbook of Human Computer Interaction. Elsevier (North-Holland), pp. 791-817.

[187] Wilson S, Markopoulos P, Pycock J & Johnson P (1992) Models in User Interface Design. In Gornostaev J (Ed.) Proceedings of the East-West International Conference on Human Computer Interaction EWHCI'92, St. Petersburg, Russia, 4-8 August, 1992, International Centre for Scientific and Technological Information, Moscow, pp. 210-217.

[188] Wilson S, Johnson P, Kelly C, Cunningham J & Markopoulos P (1993) Beyond hacking: a model based approach to user interface design. In Alty JL, Diaper D & Guest S (Eds.) People and Computers VIII, BCS HCI'93, conference proceedings, Cambridge University Press, pp. 217- 231.

[189] Wilson S & Johnson P (1996) Bridging the generation gap: From work tasks to user interface designs. In Vanderdonckt J (Ed.) Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces, CADUI'96, Presses Universitaires de Namur, pp. 77-94.

[190] Wing JM (1990) A specifier's introduction to Formal Methods. IEEE Computer, Vol. 23, No. 9, pp. 8-24.

[191] Wood CA & Gray PD (1992) User interface-application communication in the Chimera user interface management system. Software-Practice and Experience, Vol. 22, No. 1, pp. 63-84.

[192] Zave P (1996) Formal methods are research, not development. IEEE Computer, Vol. 29, No. 4, pp 26-29.

Appendix

A.1 Equivalence and pre-orders of processes

The operational semantics of LOTOS is described in two steps [18]. The first step is to interpret a LOTOS behaviour expression as a labelled transition system (LTS), by means of the axioms and inference rules associated with LOTOS syntax (summarised in table 3.2). The second step is to define equivalence classes for the derived LTSs. Different semantics are associated with the definition of the equivalence relations (and their corresponding pre-order relations). This section describes the equivalences and pre-orders used in the thesis.

A LTS is a 4-tuple $(Q, A, \xrightarrow{\mu}, q_0)$. For basic LOTOS the set A is the set of the gates of the process, plus the internal action τ, which corresponds to the explicitly specified silent action i and the implicitly specified success action δ. For full LOTOS each element of A, is a pair g<v> where g is as above, and <v> is a value description, e.g. a value identifier, etc.

Definition. Strong Bisimulation Equivalence (adapted from [118]).

A relation $R \subseteq Q \times Q$ is a bisimulation relation on Q iff $\forall (P,S) \in R, \alpha \in A \cup \{\tau\}$ the following holds:

$$P \xrightarrow{\alpha} P' \quad \Rightarrow \quad \exists S' \mid (P', S') \in R \cdot S \xrightarrow{\alpha} S'$$

$$S \xrightarrow{\alpha} S' \quad \Rightarrow \quad \exists P' \mid (P', S') \in R \cdot P \xrightarrow{\alpha} P'$$

Two processes P and S are called strong bisimulation equivalent if there exists a strong bisimulation relation R relating their initial states of their LTS, i.e. $(p_0, s_0) \in R$.

Definition. Weak Bisimulation Relation (adapted from [118]).

A relation $R \subseteq S \times S$ is a weak bisimulation relation on S iff $\forall (P,S) \in R, \alpha \in A$ the following holds:

$P \xrightarrow{\alpha} P' \quad Q'|(P',S') \quad R \cdot S \xrightarrow{\alpha} S'$

$S \xrightarrow{\alpha} S' \quad P'|(P',S') \quad R \cdot P \xrightarrow{\alpha} P'$

Two processes P and S are called *weak bisimulation equivalent*, denoted P≈S, if there exists a weak bisimulation relation R relating their initial states of their LTS, i.e. (p_0,s_0) R.

The definitions above extend to full LOTOS processes. Two LOTOS behaviour expressions B_1 and B_2, with all their free value-identifiers contained $\{x_1,..,x_n\}$ are weak (strong) bisimulation equivalent, if all instances $[E_1/x_1,..,E_n/x_n]B_1$ and $[E_1/x_1,..,E_n/x_n]B_2$ are weak (respectively strong) bisimulation equivalent where $E_1,..,E_n$, are closed value expressions of the same sort as $x_1,..,x_n$ respectively.

A *LOTOS context* C[.] is a LOTOS behaviour expression with a formal process parameter denoted by '[.]'. If C[.] is a context and B is a behaviour expression then C[B] is a behaviour expression that is a result of replacing all '[.]' occurrences in C[.] with B.

Two LOTOS behaviour expressions B_1 and B_2 are called *weak bisimulation congruent*, denoted B_1 B_2, if for all LOTOS contexts C[.], $C[B_1]\approx C[B_2]$.

Definition. φ-simulation (adapted from [118]).

Consider two LTS, $Sys_1=(Q_1,A_1,\xrightarrow{},q_{01})$ and $Sys_2=(Q_2,A_2,\xrightarrow{},q_{02})$, modelling two LOTOS processes $P_1[G]$ and $P_2[G]$.

A function $\varphi: A_1 \to A_2$, is a *coding function* from Sys_1 to Sys_2 iff

$g_1,g_2 \quad G \quad \{i,\delta\}, \quad v_1,v_2 \quad Value^* \cdot$

$g_1 \neq g_2 \quad \varphi(g_1 < v_1 >) \neq \varphi(g_2 < v_2 >)$ and $\varphi(i) = i$ and $\varphi(\delta < v >) = \delta$

A relation $R_\varphi \quad Q_1 \times Q_2$ is a φ−simulation relation for Sys_1 and Sys_2 iff

$(q_1,q_2) \quad R_\varphi, \quad \alpha_1 \quad A_1 \cdot$

if $q_1 \xrightarrow{} q_1'$ then $q_2' \quad Q_2|q_2 \xrightarrow{\varphi(\alpha_1)} q_2'$ and $(q_1',q_2') \quad R_\varphi$

Let B_1 and B_2 be two LOTOS behaviour expressions, interpreted as two LTS Sys_1 and Sys_2 respectively. B_1 is in *simulation preorder* to B_2, denoted as $B_1 \leq_\varphi B_2$, iff for some coding function φ from Sys_1 into Sys_2, there is a φ-simulation relation R_φ such that $(s_{01}, s_{02}) \quad R_\varphi$. If φ is bijective R_φ is a φ−bisimulation relation for Sys_1 and Sys_2 iff

$(q_1,q_2) \quad R_\varphi, \quad \alpha_1 \quad A_1 \cdot$

if $q_1 \xrightarrow{} q_1'$ then $q_2' \quad Q_2|q_2 \xrightarrow{\varphi(\alpha_1)} q_2'$ and $(q_1',q_2') \quad R_\varphi$

if $q_2 \xrightarrow{} q_2'$ then $q_1' \quad Q_1|q_1 \xrightarrow{\varphi^{-1}(\alpha_2)} q_1'$ and $(q_1',q_2') \quad R_\varphi$

Two processes P and S are called φ−bisimulation equivalent, denoted $P \sim_{\varphi} S$, if there exists a φ-simulation relation R_{φ} relating their initial states of their LTS, i.e. $(p_0,s_0) \quad R_{\varphi}$.

Definition. Naive transformation from full LOTOS to basic LOTOS.

The definition of the mapping P from full LOTOS behaviour expressions, to basic LOTOS expressions, is adapted from [118]. The mapping is defined here only for the LOTOS constructs used to specify ADC interactors. In this definition *g* denotes a gate, *e* denotes an ACT-ONE value expression, *x* denotes an ACT-ONE value identifier, *s* a type sort, *B* a full LOTOS behaviour expression, *Q* a LOTOS process identifier.

$P(g\ !e;\ B) = g;\ P(B)$

$P(g?x:s;\ B) = g;\ P(B)$

$P(Q[g_1,..,g_n](e_1,..,\ e_k)) = Q[g_1,..,g_n]$

$P(Q[g_1,..,g_n](x_1,..,x_k) := B) = Q[g_1,..,g_n]:=P(B)$

A.2 Graphical composition theorem for parallel and hiding operators

This section summarises a theorem concerning the regrouping and rearrangement of expressions specifying the parallel composition of processes [19]. The theorem and its proof are based on a graphical representation of parallel composition expressions, called a *process gate networks (PGN)*. A PGN is a net of nodes representing processes and nodes representing gates linked with arcs. A summary description of the theorem and the construction is included below. For a full exposition the reader is referred to [19]. The essence of the theorem is that a process gate net represents a class of strongly equivalent LOTOS behaviour expressions, containing only parallel composition operators, provided that all the process gates are explicitly shown in the net and provided they are all different.

In the following constructions Proc refers to LOTOS processes and behaviour expressions, Gates refers to the universe of gate names, and gates:Proc→Gates is a function associating to any process the set of its visible gates.

Definition. Process Gate Network.

A process gate net is a 3-tuple (P,G,E) comprising of two disjoint sets P of process nodes and G of gate nodes and a set of arcs E (P×G).

Definition. General Process Gate Net Including Hiding (GPNIH)

A GPNIH is an ordinary process gate network (P,G,E) plus a function Class:G→{I,H,V} classifying gate nodes into three categories. The gates IG={g G|Class(g)=I}are called *internal gates*, the gates CHG={g G|Class(g)=H} are

called *communicating hidden gates* and the gates CVG={g G|Class(g)=V} are called *communicating visible gates*. The gates in HG=IG CHG are called the hidden gates. The gates SG = CHG CVG are called the synchronisation gates.

Definition. Concrete Process Gate Net Including Hiding (CPGNIH)

A CPGNIH is a coherently labelled GPNIH, i.e. a 3-tuple (GN, PL, GL) where GN is a GPNIH, PL:P → Proc is a mapping from the set P to a set of LOTOS processes, GL:G → Gates is a mapping from the set G to the universe of gate-names, for which the following holds.

- GL is injective.
- Any process includes the gates corresponding to the nodes with which it is related to by the net.
- If a gate is represented in the net, and a process is not connected with the corresponding node, then it does not include such a gate in its gate set.

Definition. The set of processes (SP) of the CN

Given a CPGNIH, CN=(GN, PL, GL) the multi-set of processes SP of CN is defined as SP={PL(P_i)| P_i P}

Definition. A tree corresponding to a CPGNIH

A tree corresponding to a CPGNIH, CIN, is a tree with three types of nodes:

- Leaves, l L, over which the functions *Process* and *gates* are defined so that

  Process:L → SP(CN) is one to one.

  gates(l) = gates(Process(l)) ∩ SG(CN).

- Binary nodes, b B, corresponding to parallel composition, over which the following functions are defined assigning to each node a set of gates:

  gates(b)= gates (b.1) gates(b.2)

  syn-gates(b) = gates (b.1) ∩ gates(b.2)

The last condition is called the *maximal cooperation principle*.

- Unary nodes, u U, corresponding to hiding, over which Hidden is defined, associating to each node a set of gates, and gates is defined by

  gates(u) = gates (u.1) - Hidden(u)

The tree must verify the following conditions:

- Hidden gates are in fact hidden

  gates(root)∩HG=

- Only hidden gates are hidden

  For all hiding nodes u, Hidden(u) HG

- No communication is specified through internal gates

  For all parallel nodes p: syn-gates(p)∩IG =

- Communicating hidden gates are not hidden until all communication through them has been done.

  For all hiding nodes h: g Hidden(h) ∩ CHG · / leaf |h ≮ l and g gates(l)

  where < represents the ancestor relation between nodes of the tree.

Definition. Associated LOTOS expression.

Given a tree corresponding to some CIN, its associated LOTOS expression is defined as follows:

If t=l (a leaf) then exp(t) = Process (l)

If $t=b(t_1,t_2)$ then $exp(t) = exp(t_1)|[\text{syn-gates}(b)]|exp(t_2)$.

Theorem. Graphical Composition Theorem with Hiding

If t_1 and t_2 are two trees associated to a given CPGNIH, CIN, then the associated LOTOS expressions $exp(t_1)$ and $exp(t_2)$ are strong bisimulation equivalent.

A.3 Action based temporal logic (ACTL)

This brief introduction of the syntax and the semantics of the notation is adapted from [22]. ACTL [48] specifies states of a labelled transition system, using an action formula. The syntax and semantics for action formulae is summarised first.

Definition. Action Formulae syntax and semantics.

Given a set of observable actions Act, the language AF(Act) on Act is defined as follows:

χ ::= tt | b | ¬χ | χ χ

where b Act.

The satisfaction relation for action formulae is defined as follows:

a Δ tt always
a Δ b iff a = b
a Δ ¬χ iff not a Δ b
a Δ χ χ' iff a Δ χ a Δ χ'

The term false and the conjunction are defined as abbreviations:

ff≡¬ tt
χ χ'≡¬(¬χ ¬χ ')

Definition. The set of actions satisfying an action formula χ.

The relation κ:AF(Act) → 2^{Act} is defined as follows::

κ(tt) = Act
κ(b) = {b}
κ(¬χ) = Act/κ(χ)
κ(χ χ') = κ(χ) κ(χ')

It can be proven [22] that, for χ AF(Act), κ(χ)={a Act : a Δ χ}.

Definition. ACTL Syntax.

A state formula (denoted as φ) has the following syntax

φ::= tt | φ φ' | ¬φ | Eπ | Aπ

where A and E are path quantifiers and π is a path formula with the following syntax

π ::= $X_χ$φ | $X_τ$φ | φ$_χ$Uφ | φ$_χ$U$_{χ'}$φ

where χ,χ' AF(Act), X is the 'next' operator and U the 'until' operator.

Definition. ACTL Semantics.

s Δ tt always
s Δ φ φ' iff s Δ φ and s Δ φ'
s Δ ¬φ iff not s Δ φ
s Δ Eπ iff σ Π(s) | σ Δ π
s Δ Aπ iff σ Π(s):σ is maximal · σ Δ π
σ Δ $X_χ$φ iff |σ|≥1 and $σ_1$ → $σ_2$ | a κ(χ) and $σ_2$ Δ φ
σ Δ $X_τ$φ iff |σ|≥1 and $σ_1$ → $σ_2$ and $σ_2$ Δ φ
σ Δ φ$_χ$Uφ' iff i≥1 | $σ_i$ Δ φ' and j:1..i −1, σ(j) Δ φ and $σ_j$ $^{κ(x)}$ $σ_{j+1}$
σ Δ φ$_χ$U$_{χ'}$φ iff i≥2 | $σ_i$ Δ φ', $σ_{i-1}$ Δ φ and $σ_{i-1}$ →$^{κ(χ')}$ $σ_i$ and
j:1..i −2 · $σ_j$ $^{κ(χ)}$ $σ_{j+1}$ Δ φ

Notation. Auxiliary notation for ACTL.

- *Eventually*. EFφ stands for $E(tt_{tt}U\varphi)$ and AFφ stands for $A(tt_{tt}U\varphi)$.
- *Always.* EGφ stands for ¬ AF¬φ and AGφ stands for ¬ EF¬φ.
- ϕ *holds for a path following an action* χ. <χ> φ stands for $E(tt_{ff}U_{\chi}\varphi)$ if χ≠ff.
- ϕ *holds for all paths following an action* χ. [χ] φ stands for ¬ <χ> ¬φ.

A.4 LOTOS specifications for the decomposition example

This section of the appendix is a listing of parts of the LOTOS specification code concerning the example of decomposition of section 7.3.

```
type volume_ad is popUpSlider
opns
    inputPr:          volumeBar, Int          ->      Int
    echoPr:  volumeBar, Int             ->      volumeBar
    inputMov:         pnt, volumeBar,Int      ->      Int
    echoMov:          pnt, volumeBar, Int     ->      volumeBar
    inputRel:         volumeBar,Int           ->      Int
    echoRel:          volumeBar, Int          ->      volumeBar
    renderRV:         volumeBar, rct          ->      volumeBar
    receiveRV:        Int, rct                ->      Int
    renderV:volumeBar, Int              ->      volumeBar
    receiveV:         Int, Int                ->      Int
    result:   Int                             ->      Int

eqns
forall r:rct, p:pnt, v, n:Int, vb:volumeBar
ofsort Int
    result(v) = v;
    inputPr(vb,v) = v;
    inputMov(p,vb,v) = pntToInt(vb,p);
    inputRel(vb,v) = v;
    receiveRV(v,r)=v;
    receiveV(v,n)=n;
ofSort volumeBar
    echoPr(vb, v) = popUpSlider(vb);
    echoMov(p, vb, v) = chIconAndSlider(vb,p);
    echoRel(vb, v) = popDownSlider(vb);
    renderRV(vb,r) = changeRect(vb,r);
    renderV(vb,v) = IntToBar(vb,v);
endtype

type resizeButton_ad is pushButtonType
opns
    echoPr   :        pb_dsp, rct, pnt  ->      pb_dsp
    echoRel:          pb_dsp, rct, pnt  ->      pb_dsp
    echoMove:         pb_dsp, rct, pnt  ->      pb_dsp
    inpPr    :        pb_dsp, rct, pnt  ->      rct
```

```
    inpRel   :        pb_dsp, rct, pnt  ->      rct
    inpMov   :        pb_dsp, rct, pnt  ->      rct
    resultRB :        rct               ->      rct
eqns
forall p:pnt, ppd:pb_dsp, r:rct
ofsort pb_dsp
    echoPr(ppd,r, p) = setHilite(ppd, onH);
    echoMove(ppd,r,p) = setRect(ppd, drag(r,p));
    echoRel(ppd,r,p) = setHilite(ppd, offH);
ofsort rct
    inpPr(ppd, r, p) = r;
    inpMov(ppd,r,p) = drag(r,p);
    inpRel(ppd, r, p) = drag(r,p);
    resultRB(r) = r;
endtype
```

The ADU describes only the management of the window size and the volume.

```
process volADU[...](a:Int,dc,ds:volumeBar) : noexit :=
    doutVol!dc;        volADU[...] (a, dc, dc) []
    ainp?x:rct;        volADU[...] (receiveRV(a,x), renderRV(dc,x), ds) []
    getV?x:Int;        volADU[...] (receiveV(a,x), renderV(dc,x), ds) []
    pressVol;          volADU[...] (inputPr(ds,a), echoPr(ds,a),ds) []
    moveVol?x:pnt;     volADU[...] (inputMov(x,dc,a), echoMov(x,dc,a), ds) []
    releaseVol;        volADU[...] (inputRel(ds,a),echoRel(ds,a), ds) []
    setV!result(a);    volADU[...] (a,echoRel(ds,a), ds)
endproc

process rbADU[...](a:rct,dc,ds:pb_dsp) : noexit :=
    doutBox!dc;                rbADU[...] (a, dc, dc)[]
    pressBox?x:pnt;            rbADU[...] (inpPr(ds,a,x), echoPr(ds,a,x), ds) []
    moveBox?x:pnt;             rbADU[...] (inpMov(ds,a,x),echoMove(ds,a,x), ds) []
    releaseBox?x:pnt;          rbADU[...] (inpRel(ds,a,x), echoRel(ds,a,x), ds) []
    setMovieBox!resultRB(a); rbADU[...] (a, dc, ds)
endproc
```