Formal Aspects of Human-Computer Interaction

Gregory D. Abowd

Trinity College

June 1991

Thesis submitted for the degree of Doctor of Philosophy at the University of Oxford

Formal Aspects of Human-Computer Interaction

Gregory D. Abowd
Trinity College, Oxford

Submitted for the degree of Doctor of Philosophy, Trinity Term, 1991

Abstract

This thesis provides a constructive application of formal methods to the study of human-computer interaction. Specifically, we are interested in promoting a principled approach to the analysis and design of interactive systems that will accompany existing heuristic techniques. Previous formal approaches have concentrated on general and abstract mathematical models of interactive systems, proving that psychologically valid principles of interaction can be expressed in a language suitable for computation. These abstract models, however, are too far removed from an actual design practice which is strongly influenced by common sense and liable to break down in the face of complexity. Our efforts are focussed toward retaining the mathematical grounding of previous formalisms while providing additional insight and direction for design practice.

We introduce a unifying framework for the informal description of a user, a system and the interface that sits between them. This interaction framework provides the context for our research and motivates the properties of interaction that we wish to formalize. We adopt the view of an interactive system as a collection of agents based on the stimulus-response model. We provide a mathematical model of the agent capable of expressing interactive properties relating the goals of interaction with the visible consequences of that interaction. We also provide a language for agents which allows a natural expression of an agent's internal state-based behaviour and its external event-based behaviour. We contribute further to practical design issues by introducing templates to relate a task analysis to a specification of a system to support the tasks and an interface to adequately portray that functionality to the user. Finally, we initiate the formal investigation of multiagent architectures. This concludes the mapping of properties on abstract models of interactive systems down to properties on more implementation-based models.

Acknowledgments

I would like to thank my supervisor, Bernard Sufrin, for the initial inspiration on the topic of this thesis and for directing me to the HCI Group at the University of York for a fruitful working environment.

At York, I have had the opportunity to collaborate with Michael Harrison, Alan Dix and Russell Beale. Those collaborative efforts have resulted in several publications, parts of which have been reproduced in modified form in this thesis. The contents of this thesis represent my own work, but the thoughts and style have been greatly influenced and enhanced by contributions from these three individuals.

In addition, I have also greatly benefitted from repeated conversations with fellow students at the PRG in Oxford, students and staff at York, and colleagues on the AMODEUS project. Research in isolation has never appealed to me and I am grateful to those who have provided ample food for my thoughts and friendship for my heart.

I am particularly indebted to Janet Finlay, Roger Took, Paul Andrews, Chris Johnson and Victoria Miles for their helpful comments on drafts of this thesis.

Finally, I would like to acknowledge the support of the Rhodes Foundation which funded the first two years of my doctoral research and the European Commission which funded the final 18 months while I was working on the AMODEUS project.

Dedication

To Richard, Sara, John, Anthony, James, David, Elizabeth, Marypat, Rosemary, Michelle, Stephen, Peter, Paula, Janet, Nina, Tom, Katie, Sara, Tom, Michael, Mark. Maryclaire, David, Paul, Danny, Dennis, Michele, Peter, Kristen. Joseph, Sandy, Anne, Elizabeth, John, Philip, Paula

and

Meghan.

God created us as a family. I have felt your love as if we were separated by only an arm's length, not thousands of miles.

Contents

List of Figures

Chapter 1

Introduction

> It is well known that ninety-nine percent of the world's problems are not susceptible to solution by scientific research. It is widely believed that ninety-nine percent of scientific research is not relevant to the problems of the real world. Yet the whole achievement and promise of modern technological society rests on the minute fraction of those scientific discoveries which are both useful and true.
>
> C. A. R. Hoare in foreword to *Systematic Software Development Using VDM* [90].

> Indeed the way that people use machines is of key importance. The most significant advances in computer science will be those facilitating this interaction.
>
> From T. Winograd and F. Flores, *Understanding Computers and Cognition* [169, page 137].

The study of Human-Computer Interaction (HCI) is a multidisciplinary one aimed directly at providing scientific answers to the real problems of designing more usable computer systems. We limit ourselves in this thesis to a close examination of the relationship between two of the major contributors to HCI research—computer science and psychology. Each of these disciplines represent a respectable scientific field with concerns that subsume those of HCI. However, the main goal of HCI research is to produce methods by which the collective knowledge of psychology and computer science can be accumulated and applied to the construction of real interactive systems.

In practice, we are hard-pressed to find true interdisciplinary work significant to both fields. That which bears great psychological weight brings with it little of

practical significance to computer science, whereas that which emphasizes issues in the design of computer systems often lacks psychological validity. One reason for such a bias is that computer science and psychology are two camps which speak a language entirely unto themselves. Crucial to the symbiosis of computer science and psychology is the development of a common language between them. We are not so bold as to suggest the development of such a common language in this thesis. Rather, we hope to demonstrate how a language of design in computer science, in our case derived from the formal notations and methods familiar to software engineering, can and should be directly influenced by the vast body of psychological knowledge on usability.

HCI, as a discipline, has not reached sufficient maturity to enable software engineers to predict the usability consequences of their design decisions. One of the reasons for the lack of applicability of HCI research to real design has been noted by Harrison and Thimbleby [75]:

> We believe that much of the early work in HCI has been encumbered by a lack of appropriate abstractness or applicability to the design process.

A formal approach enables development of software whose function can be proved correct and reliable. The ease with which a human user can access this functionality is an important non-functional requirement of the software. As with other requirements on software, mathematical formulation of properties which increase usability would allow greater assurance of software usability earlier in the design process. There have been attempts in the past to show that it is possible to formulate psychologically valid properties of an interactive system in a mathematical language, but only at an abstract and general level [160, 48, 75, 161]. We take the lead from these previous examples and show that it is possible to formulate interactive properties at a more constructive and concrete level of detail which more readily reflects how the interactive system is built from a collection of cooperating objects. Hence, we maintain the ability to discuss interactive properties precisely and formally.

Some researchers have already provided a somewhat more constructive approach to design and analysis of interactive systems [14, 15, 158], but we would like to enhance their work by a stronger link to a methodology linked with empirical psychological evidence of how users perceive the tasks that computers are designed to support. To achieve this, we describe a method of design that is motivated by empirical psychological understanding of interaction. The bulk of HCI formalisms which we will describe in Chapter 3 are supported by psychological theories about how humans understand the tasks they perform on computer. The design of future interactive system must obey the lessons learned from past mistakes, and they must take advantage of increased psychological evidence to make a good product even better. In order to do that, we need a way to incorporate the psychological evidence

in the computational language of design. This cannot be achieved with the abstract and general models of interactive systems mentioned above and further discussed in Chapter 3 because those formalisms do not lend themselves well toward a construction of the interactive systems to embody the usability knowledge. We aim to provide a constructive computational formalism and method which more readily captures interactive properties and knowledge of the user at the source. The formalism we present—the agent model—provides the means for directly capturing empirical evidence, such as given by a task analysis, on interaction for some application domain.

The origins of formal methods are in computer science, so it is a fair criticism that we are biased toward the system side in our HCI research. The bias is intended, as this thesis is to be considered first and foremost as a contribution to the field of computer science, and, more specifically, to the subcomponent concerned with formal methods and its application to the design and analysis of interactive systems. Indeed, it has been precisely the success of formal methods of software engineering in general that tempts us to consider it a worthy candidate for the establishment of engineering for the user interface. Ultimate acceptance of the ideas put forth in this thesis, however, depends upon their validation by the community of psychologists active in HCI research and use by software engineers in the development and analysis of interactive systems.

There are two major contributions arising from this thesis, and they are reflected in the title. To begin with, we are interested in the promotion of formal techniques which can be used in real design. One of the major criticisms of formal techniques arise in an industrial setting where economic forces reign supreme and rigorously principled design practices do not provide ample benefit for the costs they incur. There have been isolated cases where formal development has proven an economic advantage, but for the most part, it is fair to say that formal techniques cost too much for the benefits they are *understood* to provide. We aim to show that the benefits of formal techniques have not been fully appreciated in their application toward more non-functional requirements.

This leads us to the second contribution of this thesis. We have chosen to promote the benefits of a formal approach in HCI, more specifically, in the design and analysis of interactive systems. In attempting to justify the use of formal methods for design in general, we have shown its particular advantage in HCI research where there is great need for a bridge between psychological theory and practical design, a bridge which the formal approach can provide. However, it is not only the case that formal methods can promote HCI research, for the development of the agent language in this thesis a clear case of the converse in which HCI knowledge has influenced the development of a formal approach.

When reading this thesis, it is important to keep in mind its dual purpose. We are interested in formality, but not just for formality's sake. That which we present

formally is motivated by HCI considerations. Furthermore, we are interested in HCI and the considerations which arise in the design of interactive systems, but not just for the sake of HCI. Those principles which we promote to increase the usability of a system are ones which we can capture formally within a rigorous software engineering notation. We feel this focus is necessary to ensure that our research is, as Hoare implores, both scientifically grounded and relevant.

Overview of thesis

In order to assess the multidisciplinary needs of HCI it is necessary to have an overall view of interaction that is separate from, yet sympathetic towards, both psychology and computer science. In this thesis it is hoped that some bridge may be forged between the research in both disciplines. A suitable introduction, therefore, should provide an overall view of interactive system development and analysis. We refer to this overall view as an *interaction framework*, and its description is the subject of Chapter 2.

The interaction framework provides context for assessing previous research in HCI as well as our own. There have been previous attempts at defining such context, and in Chapter 3 we will discuss these other frameworks and show how our interaction framework has been influenced by them and attempts to extend them. The interaction framework also provides a systematic way of reviewing research in HCI. In our review of Chapter 3, we will highlight the major contributions to HCI research, emphasizing how the psychological content has not fully crept into system design.

Another purpose for the interaction framework is to motivate the kinds of formalisms necessary to express properties of interactive systems. To this end, the framework naturally corresponds to an agent-based modularization of components describing the system, user and interface. Properties that affect the overall usability of an interactive system can be described qualitatively as features of the translations that occur between components of the framework. A formal model of an agent provides ammunition for a first attempt at formalizing those qualitative features of the translations. In Chapter 4, we summarize the qualitative properties of translations in the framework and formalize them in terms of a simple model of the agent, given in Chapter 2.

The simple agent model is inadequate for two reasons. First, it is not rich enough to express interactive properties which relate the goals or end results of interaction to the more immediate perceivable information provided at the interface. Psychological formalisms essentially relate tasks that a system should support to the results that those tasks affect. The simple agent model's inability

to capture constructively result information prevents its use in an overall psychological/computational design method. Secondly, it is not a design notation. We cannot use the simple agent model to describe complex systems by the composition of smaller and simpler subcomponents.

In order to address these two inadequacies, we digress from the specific application of HCI to concentrate attention in Chapters 5 and 6 on the development of a refined agent model and associated language. Chapter 5 defines the new agent model in terms of three perspectives—the internal, state-based specification of the agent, its external event-based specificatiou and a communication specification which links internal operations to external events. The refined model is shown to obey essential compositional properties to allow for a modular design approach. In Chapter 6 we justify the need for a new formal language to describe agents, which is a hybrid notation combining a model-oriented notation similar to Z or VDM and a process algebra notation similar to CSP or CCS, capitalizing on the familiarity and expressiveness of each.

Armed with greater detail of the structure of an agent and a language for describing agents, we resume in Chapter 7 with the formal treatmeut of interactive system design. The concentration in this chapter is toward showing how notions of result and display allow salient description of interactive properties. In addition, the introduction of templates allows for a method of design specifically geared towards the immediate incorporation of psychological evideuce in interactive system descriptions. Result templates embody task analytic information which guide the initial descriptiou of a system's functional core. Display templates are chosen to correspond to the result templates and satisfy some interactive properties, such as predictability, honesty, consistency and others.

It is easier to formally express interactive properties when we remain at the abstract and general level. We are then able to ignore the clutter of implementation detail which is not entirely relevant to the expression of the properties. However, we cannot ignore the inevitable refinement towards executable systems, and so we must consider interactive system architectures and how the abstract principles and properties can be mapped into a more realistic design platform. Iu Chapter 8 we describe the relationship hetween the abstract level of the interaction framework and the more concrete multiagent architectures that bave been used to describe the structure of an interactive implementation. The formal agent model allows us to describe more precisely the features of the heuristically-based multiagent architectnres and the design methods they imply iu order to assess their value for preserving the properties expressed at the abstract level.

We conclude in Chapter 9 with a summary of the results and contributions of this thesis, aloug with an agenda for future research.

In addition to the main body of this thesis, we provide three appendices. We make extensive use of the Z notation throughout the thesis, and we will assume familianty with the standard notation as provided by Spivey [152]. However, for stylistic reasons we have deviated from the standard nse in some situations. In addition, it is often the case that we will need to introduce some special notation to make the expression of some predicate more concise. In order not to detract from the flow in the main body of the thesis, in Appendix A we have described our stylistic conventions which deviate from standard Z, along with any notation that is not defined in [152]. Chapter 5 contains the most concentrated sections of formalism in the thesis. We have relegated to Appendix B the proofs of some theorems on the refined agent model, leaving only the outline of their proofs in the thesis body. Finally, in Appendix C we provide greater detail on the semantics for the agent language in terms of the agent model than was deemed appropriate for the body of Chapter 6.

Chapter 2

A Framework for Discussing Interaction

We present a general interaction framework which will allow the analysis of the interactiou between a user and a system to be expressed in one, unified language. Our intent in this thesis is that the system be some computerized application, but this assumption does not affect the iuteraction framework. It is also a common interpretation that by distinguishing betweeu user and system we are restricted to single-user applications. This restriction is not an underlying assumption in development of the framework, but rather results from one's interpretatiou of what system and user represent. The emphasis in the framework is in developing a view of interaction from a single user's perspective. In a multiple user application, such as a multi-party confereuciug system, from the point of view of any one user the rest of the users form part of the system.

The interaction framework will be used as a bridging device to provide a common ground for both psychological and computational discussion of interaction and interactive systems. It is important that it be understood at some level by both psychologists and computer scieutists. Therefore, it must be free from the jargon of both fields and open to accurate interpretation based on common sense. By making common sense principles explicit within the model we open the path to their automatic inclusion in future design.

Overview of chapter

This chapter proceeds with an informal description of the major components and translations in the iuteraction framework in Sectiou 2.1. A simple stimulus-response model of an agent will provide a formal model for the framework in Section 2.2. Both the informal and formal agent descriptious will be used to express properties of interactive systems which attempt to qualify and quantify usability throughout the remainder of this thesis.

2.1 An informal definition

The purpose of an interactive system is to aid a user in accomplishing *goals* from some application *domain*. A domain defines an area of expertise and knowledge in some real-world activity. Some examples of domains are graphic design, authoring and process control in a factory. A domain consists of concepts which highlight its important aspects. In a graphic design domain, some of the important concepts are geometric shapes, a drawing surface and a drawing utensil. *Tasks* are operations to manipulate the concepts of a domain. A goal is the desired output from a performed task. For example, one task within the graphic design domain is the construction of a specific geometric shape with particular attributes on the drawing surface. A related goal would be to produce a solid red triangle centred on the canvas.

Our definitions of goal, task and domain generally agree with the approach to problem solving presented by Newell and Simon [119]. We will apply the general term *task analysis* for the identification of the problem space for the user of an interactive system in terms of the domain, goals and tasks. In Chapter 7, we commit to an even more rigid definition of task analysis as a mapping from tasks in the user's goal structure to a set of features or attributes that are intended to represent the action of that task in the system.

The identification of goals and tasks in a problem space is crucial to the work of most analytic approaches to HCI, including our own, for they determine the starting point for analysis of a design, as Lewis points out [101]. There are those who object to the use of goals and tasks as fixed starting points for analysis. Whiteside and Wixon [166] argue that their inclusion should only be as dynamic reflections of broader environmental issues in HCI and not as static entities from which all analysis can proceed with sound grounding. Carroll agrees, pointing out that the usefulness of task analysis to designers is minimized without due attention to such contextual information, usually lacking in most theoretical approaches [35]. A more drastic opinion is put forth by Suchman [157], who states that an understanding of social interaction, not found in existing task analysis techniques, should be the driving force for any theory of single user HCI.

Our belief in the context of this thesis is that an awareness of the goals and tasks of a user in a particular domain form a crucial guide to the assessment of the computer system designed to support the interaction. Therefore, we are not as concerned with the overall theory of cognition and social environment which identifies the goals and tasks as we are concerned with how a definition of task can be used to aid the formal development of the computer system.

Typically, the concepts used in the design of the system and the description of the user are separate; they are considered separate components, and so we refer to them separately as the *System* and the *User*, respectively. The *System* and *User* each have a domain-specific language in which the concepts can be expressed.

These languages treat both *System* and *User* as state machines with operations that can transform the underlying state. The *System*'s language we will refer to as the *core language* and the *User*'s language we will refer to as the *task language*.

The core language contains system attributes, describing concepts in the domain relevant to the *System* state. The task language contains psychological attributes, describing concepts in the domain relevant to the *User* state. At the most abstract level, both system and psychological attributes are not constrained by any implementation details. Within a formal approach, all that is required is that these attributes be represented by some mathematical object. And once the domain has been adequately captured mathematically, it can be manipulated and reasoned about with rigour.

At the formal and abstract level it is possible to attain a close correspondence between the descriptions of the *System* and *User*, especially in a user-centred design practice which uses the psychological attributes to determine the system attributes of interest. This is the purpose for a task analysis method—to produce some description of the user's understanding of the domain so that a tool can be properly designed for work in the domain. Though there may be a close correspondence between the system and psychological attributes, the *User* does not directly interact with the system attributes. Rather, the interaction is with a representation of the system attributes that is constrained to a far less expressive language of the *physical interface*, exemplified for the most part by two-dimensional displays with primitive sound features and limited tactile facilities.

2.1.1 The interactive cycle

The communication between the *User* and the *System* follows a cycle of execution and evaluation, as explained by Norman [123, 124, 125]. The *User* formulates a goal and then must decide the task to perform in order to achieve the goal. The task is executed upon the *System* and the result of the operation is evaluated to see if it agrees with the original goal. This gives four main phases to the interaction—formulation, execution, evaluation and assessment, as shown in Figure 2.1. We will further discuss the execution/evaluation cycle of interaction in the context of previous HCI research in Chapter 3.

Since the result of user-centred design as described above is that the task language of the *User* and the core language of the *System* are closely related, interaction between *User* and *System* is fairly straightforward, since the translation between the two languages can and should be trivial. The simplicity of interaction implied by this close correspondence between the abstract *System* and *User*—what the *User* wants to do the *System* can do—is misleading, because it is often the case that there is a mismatch between the *User*'s high-level task language and the low-level entities of the physical interface which the *User* must manipulate in order

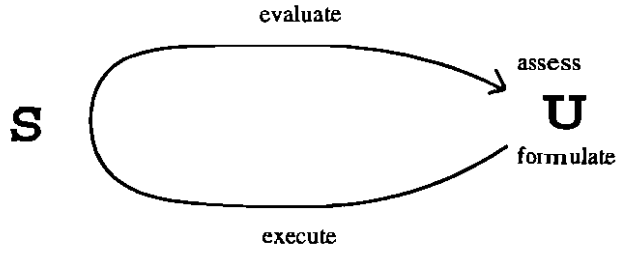

Figure 2.1: Four phases of interaction between *User* (U) and *System* (S)

to achieve the desired goal. There is also a mismatch between the task langnage and that which the *User* interprets from observations of the physical interface when determining if the goal has been achieved. These two mismatches are referred to, respectively, as the gaps of execution and evaluation by Hntchins, Hollan and Norman [85].

2.1.2 The components of the framework

In order to attain a more realistic description of interaction, therefore, we break down the interaction between user and machine into four main components, as shown in Figure 2.2. The nodes represent the four major components in an inter-

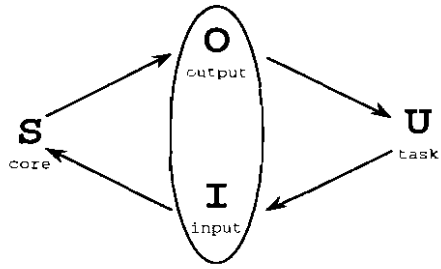

Figure 2.2: The general interaction framework

active system—the *System*, the *User*, the *Input* and the *Output*. Each component has its own language which is used to express its purpose in the interaction. In addition to the *User*'s task language and the *System*'s core language which we have already introduced, there are languages for both the *Input* and *Output* components

to represent those separate, though possibly overlapping components. *Input* and *Output* together form the system interface. Note that we distinguish between the system interface and the physical interface. The physical interface is that part of the system which, as its name suggests, is in direct contact with the user in the physical world. Therefore, the physical interface is viewed as a subset of the interface in our framework. The input and output languages do not in most cases map very directly onto concepts in the domain. Yet, the interface's position between *System* and *User* mandates that it be an effective mediator for the tasks in the domain of the application. Therein lies the major challenge in interactive system design.

As the interface sits between the *User* and the *System*, there are four steps in the interactive cycle, each corresponding to a translation from one component to another, as shown by the labelled arcs in Figure 2.3. The *User* begins the

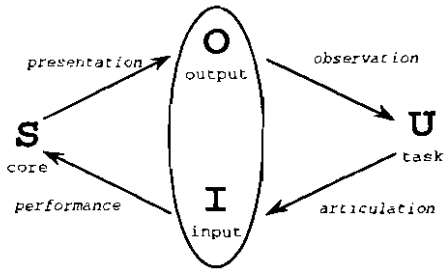

Figure 2.3: Translations between components

interactive cycle with the formulation of a goal and task to achieve that goal. The only way way the user can manipulate the machine is through the *Input*, and so the task must be articulated within the input language. The input language is translated into the core language as operations to be performed by the *System*. The *System* then transforms itself as described by the operation translated from the *Input*; the execution phase of the cycle is complete and the evaluation phase now begins. The *System* is in a new state, which must now be communicated to the *User*. The current values of system attributes are rendered as concepts or features of the *Output*. It is then up to the *User* to observe the *Output* and assess the results of the interaction relative to the original goal, ending the evaluation phase and, hence, interactive cycle.

It is easiest to think of the interactive cycle as a true alternation between execution and evaluation, but this is not always the case. Every action by the *User* may not be followed by an evaluation, and it is very possible that the *User* will be required to observe changes to the *System* that were not directly prompted by

actions performed by the *User*. Therefore, a strict interpretation of the interaction framework in terms of an alternating execution/evaluation cycle is not intended in our presentation. However, for explanatory purposes, this interpretation is not overly harmful.

2.2 A formal definition

In the previous section, we presented an informal introduction to the interaction framework. In this section, we will provide a formal definition of the framework. The formal model of the interaction framework provides the foundation for the work of this thesis, motivating the formulation of abstract properties of interaction in Chapter 4, the agent model of Chapter 5 and the more constructive interactive properties of Chapter 7. We view an interactive system as a collection of communicating interactive *agents*. This section proceeds by giving a simple formal definition of an agent and a description of each of the components in the interaction framework in terms of agents. We then combine the different agents for a description of a complete interactive system as suggested by the interaction framework and translations between components suggested by Figure 2.3.

2.2.1 A simple definition of an agent

The description of an agent serves two purposes—to give its current state and to describe how that state is transformed as its interaction with other agents proceeds. We make a distinction, therefore, between the *state* of the agent and the *events* in which it participates in cooperating with other agents. We present two given sets to represent the set of all possible states of an agent, *State*, and the set of all possible events an agent can participate in, *Event*.

$$[State, Event]$$

Further details of the state and event sets will be delayed until the refinement of the agent model described in Chapter 5.

The link between the state of an agent and the events in which it participates is given by the agent's *behaviour*. An agent is based on the stimulus-response model, which has been argued to form part of the user's and designer's model of an interactive system [41]. An agent participates in a stimulus event which triggers a change in the internal state. After the transition, the agent responds with events which will affect other agents in the system. From this description, we choose to model the agent's behaviour as a relation between stimulus-state pairs and response-state pairs. In a given state, receipt of a single stimulus will result in a new state and a sequence of observed response events. Two views of this behaviour

concentrate on differents aspects of it. One view gives the state transformation triggered by a stimulus event. The other view gives the pairing between stimuli and responses. This stimulus-response information is an external description of an agent, whereas the state transformation mapping gives internal information on the agent. The simple formal agent model is given by the schema *Agent* below

$$
\begin{array}{|l}
\textit{Agent} \\
\hline
\textit{states} : \mathbb{P}\, \textit{State} \\
\textit{inits} : \mathbb{P}\, \textit{states} \\
\mathcal{B} : (\textit{Event} \times \textit{states}) \leftrightarrow (\mathrm{seq}\, \textit{Event} \times \textit{states}) \\
\textit{transform} : \textit{Event} \nrightarrow (\textit{states} \leftrightarrow \textit{states}) \\
\textit{stimresp} : \textit{Event} \leftrightarrow \mathrm{seq}\, \textit{Event} \\
\hline
\forall\, \textit{stim} : \textit{Event};\ \textit{resp} : \mathrm{seq}_1\, \textit{Event};\ s, s' : \textit{states} \\
\quad \bullet\ ((\textit{stim}, s), (\textit{resp}, s')) \in \mathcal{B} \Leftrightarrow (\ (\textit{stim}, (s, s')) \in \textit{transform} \\
\qquad\qquad \wedge\ (\textit{stim}, \textit{resp}) \in \textit{stimresp}) \\
\hline
\end{array}
$$

We can define two interpretation relations on seqnences of stimulus events, or *programs*. The first, I^{int}, gives the internal interpretation of the program, i.e., the possible state that the agent can be in after participating in the program. Each stimulus event in the program represents a state transition relation, and so the overall state transition relation for the program is the sequential composition of the individual state transition relations. Restricting the domain of this overall transition relation to the initial states of the agent ($A.inits$) gives the possible transitions for the agent.

$$
\begin{array}{|l}
\textit{transformExtend}\_ : \textit{Agent} \nrightarrow \mathrm{seq}\, \textit{Event} \nrightarrow (\textit{State} \leftrightarrow \textit{State}) \\
\hline
\textit{transformExtend}_A(\textit{stims}) = (A.\textit{inits}) \lhd (\fatsemi/(\textit{stims} \fatsemi A.\textit{transform})) \\
\hline
\end{array}
$$

The internal interpretation function relates the program to the possible final states.

$$
\begin{array}{|l}
I^{int}\_ : \textit{Agent} \nrightarrow \mathrm{seq}\, \textit{Event} \leftrightarrow \textit{State} \\
\hline
(\textit{stims}, s) \in I^{int}_A \Leftrightarrow \\
\qquad \exists\, s_0 : A.\textit{inits} \bullet (\textit{stims}, (s_0, s)) \in \textit{transformExtend}_A \\
\hline
\end{array}
$$

After each input event, the agent responds with a sequence of response events, as given by *stimresp*. A program of inputs, *prog*, is therefore related to a sequence of responses, *respobs*, derived from the responses of each input event in *prog*. The second interpretation relation, I^{ext}, provides this derived overall stimulus-response behaviour.

$$
\begin{array}{|l}
I^{ext}_{\_} : Agent \rightarrow \mathrm{seq}\, Event \leftrightarrow \mathrm{seq}\, Event \\
\hline
\mathrm{dom}\, I^{ext}_{A} = \mathrm{dom}\, I^{int}_{a} \\
(prog, respobs) \in I^{ext}_{A} \Leftrightarrow \\
\quad \exists\, respseq : \mathrm{seq}\, \mathrm{seq}\, Event \\
\quad \mid \quad (\quad \#respseq = \#prog \\
\qquad \wedge \frown\!/\, respseq = respobs) \\
\quad \bullet \ \forall\, i : 1..\#prog \\
\qquad \bullet \ (prog(i), respseq(i)) \in A.stimresp)
\end{array}
$$

Note that captured in this definition is the assumption that all of the responses for a given stimulus event are observed before any responses owing to snbsequent stimuli. This assumption is not mandatory. We could have replaced the predicate $\frown\!/\, respseq = respobs$ with one such as $(respseq, respobs) \in interleaves$, with *interleaves* as defined in Appendix A, or even more sophisticated expressions, to reflect the more general possibility that the order of responses does not fully respect the order of the stimuli. We will disregard that possibility in this thesis, for it is believed to bring more tedium than enlightenment.

2.2.2 An agent description of an interactive system

Onr iuitial understanding of an interactive system suggests that it is composed of two agents, representing the *User* and the *System*. In addition, there are translations between the stimuli of one and the responses of the other, which represent the externalized aspects of the execution and evaluation phases. These translations are formalized as relations between the stimnli and responses of the agents. The *execution* translation is a relation between the responses of the *User* agent and the stimuli of the *System* agent. The *evaluation* translation is a relation between the responses of the *System* agent and the stimuli of the *User* agent. A simple interactive system is defined below in terms of the *User* and *System* agents and the translations between them.

$$
\begin{array}{|l}
\textit{SimpleIntSys} \\
\hline
User, System : Agent \\
execution, evaluation : \mathrm{seq}\, Event \leftrightarrow \mathrm{seq}\, Event \\
\hline
execution \in (\mathrm{ran}\, I^{ext}_{User} \leftrightarrow \mathrm{dom}\, I^{ext}_{System}) \\
evaluation \in (\mathrm{ran}\, I^{ext}_{System} \leftrightarrow \mathrm{dom}\, I^{ext}_{User})
\end{array}
$$

The translations *execution* and *evaluation* are relations between event sequences, similar to the external interpretation relation for an agent. We can therefore regard *execution* and *evaluation* as specifications of the overall stimulus-response behaviour of agents between *System* and *User*. Constraints on these translations will

be constraints that must be satisfied by the agent which manifests that specification. Therefore, the description of a simple interactive system contains information on four agents, two explicit (*User* and *System*) and two implicit (*execution* and *evaluation*).

The complete interaction framework builds from this simple definition by adding the *Input* and *Output* agents along with translation relations labelled as those in Figure 2.3. The *articulation* and *performance* relations are composed to yield the *execution* translation and the *presentation* and *observation* relations are composed to yield the *evaluation* translation. As was the case for *execution* and *evaluation*, these translations provide specifications for additional implicit agents in the system.

InteractionFramework

$$
\begin{array}{l}
SimpleIntSys \\
Input, Output : Agent \\
articulation, performance, \\
presentation, observation : \text{seq } Event \leftrightarrow \text{seq } Event \\
\hline
execution = articulation \fatsemi I^{ext}_{Input} \fatsemi performance \\
evaluation = presentation \fatsemi I^{ext}_{Output} \fatsemi observation \\
articulation \in (\text{ran } I^{ext}_{User} \leftrightarrow \text{dom } I^{ext}_{Input}) \\
performance \in (\text{ran } I^{ext}_{Input} \leftrightarrow \text{dom } I^{ext}_{System}) \\
presentation \in (\text{ran } I^{ext}_{System} \leftrightarrow \text{dom } I^{ext}_{Output}) \\
observation \in (\text{ran } I^{ext}_{Output} \leftrightarrow \text{dom } I^{ext}_{User})
\end{array}
$$

2.3 Conclusions

From our above discussion, we can see that the interaction framework contains at least partial information on eight different agents—the four major components of the *User*, *Input*, *System* and *Output*, and external specifications on four agents which represent the translations between the stimuli and responses of the major components. Since agents are intended as a compositional and constructive model for an interactive system, the actual agent description of a complete interactive system will contain many separate agents. Our point in this thesis is that we can view everything in the interactive system as an agent. Therefore, properties of interaction which we will present in the remainder of this chapter and throughout the thesis can be expressed as properties on agents.

The justification for our division of an interactive system into four major components and four translations between them is the subject of the next chapter, in

which we will use the framework to establish the context of previous HCI research and establish the further contributions of this thesis for HCI.

Chapter 3

Background and Related Work

The last chapter provides context for the application of formal methods in HCI by describing a general interaction framework. In this chapter, we will review previous research into establishing the general context of HCI research and formalizing different aspects of interaction. Before launching into the review, we will highlight two of the major conclusions which the review supports.

The first conclusion is that a unified framework, such as presented in Chapter 2, helps determine what psychological information is available to feed into design. Though there is a basic divide between the precision with which reasoning is possible on the two sides of interaction—the human (or user) and the computer (or system)—it is valuable and instructive to view both within the same formal (and informal) framework. The system side deals with objects that can be quantified and reasoned about mathematically. This is the premise upon which the application of formal methods in software engineering is based. The system attributes mentioned in Chapter 2 which are used to describe domain concepts are intended to have executable realizations in refined versions of the system. They may initially be presented as abstract concepts, but the whole purpose of refinement work is to realize an abstraction in concrete detail while preserving the properties of the abstraction.

On the user side, the psychological attribntes represent attempts to describe phenomena whose very existence is itself a research question. Though several formalisms exist which provide quantitative predictive power for analysis of the user's side of interaction, the conclusions they provide are questionable from both a psychological and design perspective. However, empirical and theoretical psychological evidence is able to support some assumptions that we can make about the human as user which we can then incorporate into our design process. Incorporating these assumptions about the user explicitly in the design process allows their removal or alteration if they are found to be invalid.

From the design perspective, there is a crucial symbiotic relationship between

system and psychological formalisms. The empirical psychological evidence for interactive behaviour provides data which the system formalism can manipulate. The interactive design method reqnires both formalisms; hence the need for one framework which unifies the two.

A second conclusion supported is that the bridge between psychological and computer science research is not heavily travelled by researchers in formal methods and software engineering. Though the vast majority of software developed is interactive and could therefore be aided by a rigorous theory of interactive system design, very few designers are engaged in such principled design. The majority of literature in software engineering that covers interactive system development relies on heuristic reasoning about good design. Though a major reason for this is the lack of acceptance of formal notations in general in design practice [44], there is little work done under the name of formal methods which gives fair notice to the consequences of including user considerations in design or specification. We see two reasons for this. First, most of the formalisms which have been offered come from researchers who are mainly psychologists and not, therefore, concerned directly with the design implications of their formalisms. Second, many of the formal techniques availahle and in nse in industry do not provide enongh descriptive power to naturally express an interactive system in the way the designer (and the user) perceives it. This last topic is a major consideration in our development of the agent model and its associated specification langnage in Chapters 5 and 6.

As a result of these conclusions, we can see the pnrpose of this thesis. We aim to provide a theory of interactive system design that both makes its psychological assumptions explicit for means of validation and is within the grasp of the software engineering profession (or at least practitioners of formal methods within software engineering). By building on and extending previous research in both HCI and formal methods, we present a method for interactive system design which unifies previously separated considerations abont the user, the system and the interface which separates them.

Overview of chapter

In Section 3.1, we will relate the interaction framework of Chapter 2 to previous attempts to define the context of HCI research. Our interaction framework is not the first attempt at breaking up the interaction between a user and a computer into stages. We will present some other frameworks for interaction that predate and influence our own, explaining how our interaction framework extends their work. In Section 3.2, we will present a survey of the research applied to various aspects of HCI. In this section, we will describe in more detail some of the formal and informal research which presents a more narrow focus than the frameworks discussed in Section 3.1. We have tried to classify the different approaches according

to how they fit into our interaction framework.

3.1 Other interaction frameworks

We identify four major categories of general and informal theories that have inspired our framework, and we will discuss each in this section. The purpose of frameworks, such as ours, is "not to reveal dramatic new truths," as Norman points out [123]. Rather, the purpose is to provide insight into the implications each stage within the framework has on the design of interactive systems.

We will emphasize how each of the frameworks below compares to our framework. We hope this serves as a suitable justification for the introduction of yet another view of interaction which will guide the formal approach of the remainder of this thesis.

3.1.1 Evaluation/Execution Cycle

Probably the most obvious influence on our interaction framework has come from the execution/evaluation cycle of interaction. This view of interaction is made explicit in much of the literature on HCI and it is implicit in nearly everyone's common sense nnderstanding of the interaction between human and computer. The human user formulates a plan of action which is then executed at the computer interface. Upon completing the execution of some plan, or part thereof, the user observes the compnter interface to evaluate the result of the recently executed plan and to determine the further course of action. In Chapter 2, we acknowledged the seminal work of Norman [124]. A similar division of the interaction cycle has been made explicit hy Card, Moran and Newell [32], and we will examine their contribution further in Section 3.2 as it is more detailed in the formalism which it presents. The work by Norman which we reference is mainly qnalitative, and so most resembles the presentation of our interaction framework. His views of the iuteraction between user and computer have been criticized as too simplistic, but we view the real value of his views in the direct appeal to common sense. Arguments that it over-simplifies a complex topic fail to see its purpose as a readily understandable overview of human-computer interaction accessible to those lacking a formal psychological education. Cousequently, Norman's model is accessible to the computer scientist interested in designing a more usable system. Unfortunately, Norman's model does not consider the system's contribution to the execution and evaluation cycle as much as the user's coutribution. Our interaction framework is intended to address this disparity.

Norman initially ontlined four stages of the user's activities—intention, selection, execution and evaluation [123]. In later work [124, 125, 85], the interactive cycle can be seen as divided into two major phases, execution and evaluation. Each

of these phases is then subdivided further into different stages of the interaction that can be examined for their particnlar influence on the effectiveness of the overall interaction. The seven stages mentioned are [124, page 41]:

1. Establishing the Goal
2. Forming the Intention
3. Specifying the Action Seqnence
4. Executing the Action
5. Perceiviug the System State
6. Interpreting the State
7. Evaluating the System State with respect to the Goals and Intentions

These stages are further related by Norman. as he symmetrically divides the previous execution/evaluation cycle. Hence, perceiving the system state is seen to be the evaluative equivalent of executing the action, interpreting the state is the evaluative equivalent of specifying the action sequence and evaluating the system state with respect to the goals and intentions is the evaluative equivalent of forming the intention.

Figure 3.1 portrays the relationship hetween the execution/evaluation cycle and our interaction framework. The obvious comment, as we have already mentioned, is that the execution/evaluation cycle does not consider the system beyond its interface (hence the shaded left-hand side in Figure 3.1). Norman simply represents the system as the world of physical activity, analysis of which stops at the physical interface. There is much greater detail on the user's side of the interface, as the translation stages from *User* to *Input* (articulation) and from *Output* to *User* (observation) in our framework are each further divided into three substages in Norman's model.

Norman's model serves two purposes. It provides the first step needed to introduce a computer scientist to the purpose of psychological work in assessing the usability of interactive systems. It also provides an outline for previous and future theoretical or empirical research by psychologists trying to describe how the user interacts with the system and how that interaction can be assessed. This interaction is divided into two phases. One is concerned with the formulation of a plan and its performance. The other phase is concerned with the observation of the results of previously performed plans and their assessment with respect to the original plan. As a rule, formal psychological research into human-computer interaction, therefore, is divided roughly into two areas, one to address the cognitive aspects of formulation and execution of the plan and one to address the cognitive aspects of

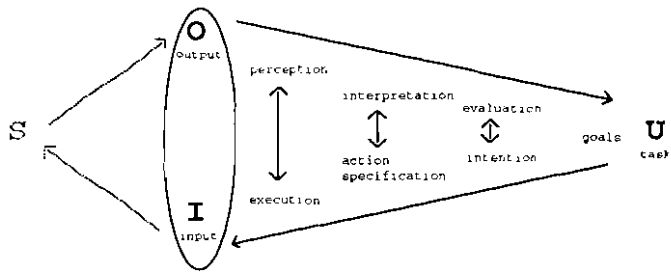

Figure 3.1: Relationship between Norman's execution/evaluation cycle and the interaction framework

perceiving and assessing the consequences of the executed plan [30]. One notable exception to this rule is found in the research by Barnard on Interacting Cognitive Subsystems (ICS) [17, 18], which is a model of human cognition and performance that can address both evaluation and execution in the context of human-computer interaction.

Our interaction framework, therefore, is an attempt to extend Norman's model with a necessary and complementary component which more fully addresses issues of the system. The extension provides a first step needed to introduce a psychologist to the needs of computer science in designing interactive systems. In addition it provides a platform for assessing previous system-based work on interactive systems and provides direction for future formal research on interactive system design.

Norman's lucid account of the execution/evaluation cycle provides a clear insight into what factors affect the translation between languages between the *User* and the interface (*Input* and *Output*). We will discuss in Section 3.2 how Norman's qualitative account can assist in assessment of more formal accounts of these translations. The addition of the *System* and its relationship to the interface in our interaction framework is viewed as a necessary and complementary view that enhances Norman's model. With proper attention focused on the system side, we can also assess previous work on interactive system development and propose how a formal approach can better aid the precise development of more usable systems.

3.1.2 Interaction Modelling Framework

Barnard and Harrison [20, 19] have also presented an interaction framework but for a different purpose than either our framework or the execution/evaluation model. The purpose of their framework is to direct research on incorporating distinct models of the system and user, which already exist, by means of a separate and new interaction model. It is for this reason that we distingnish this framework from

ours by referring to the Barnard and Harrison version as an interaction modelling framework. Figure 3.2 is a graphical representation of their framework, taken from [19].

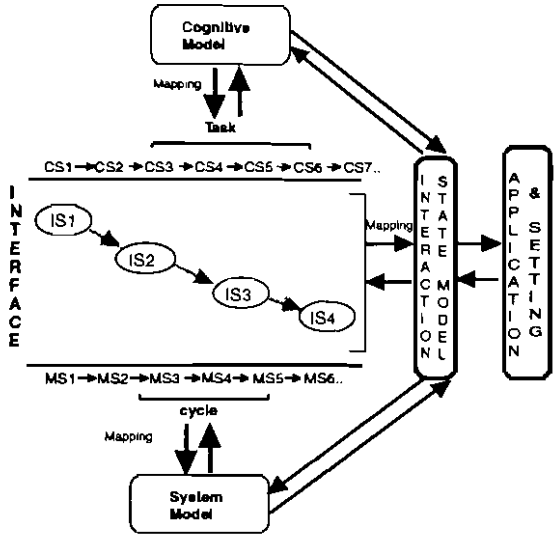

Figure 3.2: Barnard and Harrison's interaction framework.

Barnard and Harrison identify some major problems of system and user models of interactive behaviour. System models make implicit psychological assumptions about the user that can be invalid. For example, the PIE model version of predictability [54, 48], which we will discuss later, makes the assnmption that a system designer can *a priori* determine what effects will be perceived and understood by the user in terms of their task langnage. User models make implicit assumptions about the practice of system design that are impractical. For example, a psychologically valid notion of consistency of an input languages is presented by Payne and Green's TAG notation [127, 128], but their device representation does not consider that system design must take into account more than just inpnt langnage consistency. It is not so much a problem that the separate modelling domains make assumptions about the other. The problem is when the modelling technique depends on implicit assumptions that it cannot change. In an ideal situation, each modelling domain (via the modellers) must be able to validate the assumptions made by the other modelling domain and also incorporate the results of the other domain within its

own work. How can this be achieved?

In Barnard and Harrison's view, there is a common feature that can be extracted from both models of the system and models of the user. Each can describe its subject in terms of a state-based machine that undergoes transitions. The system states are *machine states*, and the user states are *cognitive states*. Any instance of interaction between user and system results in a sequence of transitions in both the machine and the user. Figure 3.2 depicts these transitions by the sequence of machine states $MS1, MS2, \ldots$ and cognitive states $CS1, CS2, \ldots$ There is no clear link made between machine state MSi and cognitive state CSi. The intention of the mappings from models to state transitions in Figure 3.2 is to show that the system and user models are not wholly determiued by what they say about state transitions. Rather, those models each say something about the behaviour of the machine or cognitive state of the user that can be mapped onto state transitions. In turn, particular instances of the machine or user behaviour can be mapped into the respective model for interpretation relative to abstractions within the model, as we will discuss next.

Within a system or user model abstractions can be defined in terms of the transitions of the machine or cognitive state. For example, there is the notion of cycle within the system model and the notion of task in the user model. In trying to establisb the relationship between the abstractions made by the system modellers and those of the user modellers. they suggest a third model, called the interaction model. The reasoning is that any relationship between system and user modelling concepts would only be relevant where the two models meet, that is, at the human-compnter interface. Therefore, a model of the interface would be the suitable location for the cross-fertilization of the different concepts of system and user modelling withont undue emphasis on eitber side of the interface. This interaction model would also have an interpretation based on states, labelled by 'interaction states', and iustances of transitions wonld be labelled $IS1, IS2, \ldots$, as in Figure 3.2. The interaction model is fed by information about system and user models but in addition can also embody explicitly notions that are usually lacking witbin the system and user models, such as the domain knowledge.

Finlay *et al.* have attempted to apply the interaction modelling framework to the analysis of exemplar systems, such as an automated teller machine, by making explicit an event structure used to link the system and cognitive models [57]. The significance of the events used hy Finlay *et al.* is that they can both be handled by the respective system and cognitive modellers. Events are labels for operations or state transitions. At the interface between user and computer, the same event can be linked to operations on both the user and the system. On the system side, events are similar to those described in Chapter 2 for the simple formal model of an agent. On the user side, Barnard's own ICS model is used, and there is a much less convincing connection between events and the ICS way of modelling

the user's cognitive states and operations performed upon them. This points up not a theoretical failing of ICS but rather its inability in its present form of being expressed in a formal notation similar to that used on the system side. More recent work by Barnard and Harrison [22] has concentrated on providing a uniform description of both user and system model structures.

There is a stronger similarity between the interaction modelling framework and the refinement of our own framework introduced in Section 2.2. In that section, we begin to formalize the concepts of the interaction framework. Our refined interaction framework will address directly the problem discussed above of the linking work done by Fiulay *et al.*, because we will provide a common notation for both the user and system models. What would remain is to express a cognitive model such as ICS within that notation.

3.1.3 Black box models

A major difference between how an interactive system is viewed by a designer as opposed to a user is that the designer knows about the intricate detail of the computer system and the user does not. A result of this knowledge deficiency is that the user attempts to formulate a model of how the system works. This model is in part determined by the experience the user has in interacting with the system. In these interactions, all that a user will know about the system will come from all of the perceivable information which the system presents. A principle of user-centred design is that the designer try to match the user's perception of the interaction. This principle justifies to some extent why psychologists focus their analysis on only that part of the system which is observable, since that is the only information that the system provides the user. But in attempting to think as the user, it is often hard for the designer to be divorced from the intricate system detail.

This knowledge discrepancy causes a problem with the principles of user-centred design because the designer knows too much about how a system functions in order to objectively judge such properties as predictability as the user would perceive them. One solution is to analyze the system by forgetting (temporarily) the internal details. Essentially, a user views the computer system as a *black box*, noticing only that which is presented as input to the system and that which is produced as the effect inspired by the input. If the designer also adopted this black box view of the system, then a fairer assessment of the interactive properties of the system could then proceed.

A more important use of a black box model would precede a full specification and implementation. In this case general domain-independent principles can be used to guide the development. Dix *et al.*[52] show how such principles can be used to guide the design of a small programming environment. Monk and Dix [113] use a semi-formal black box model to examine how action-effect rules can provide insight

iuto the application of general principles of predictability, simplicity, consistency and reversibility within design.

As an aside, these approaches to specification and design suggest a revised software lifecycle, as presented by Harrison [71], in which a model of interaction, of which a black box model is but one example, is used as a link from the requirements phase to the specification phase.

requirements → *interaction model* → *speeification* → *implementation*

The uotion of a black box is quite a common one in software engineering and system development; it appears under the name of data abstraction and the abstract data type [102] or information hiding [126]. It is also quite common in other sciences; for example, the transfer function in control theory. The classic example of a black box approach to interactive system analysis is the *PIE model*, first presented by Dix and Runciman [54], and later greatly expounded upon by Dix [48, 49]. The model cousiders user input to the system as coming from a set of programs P and output as being from a set of effects E. The system is modelled as a black box whose entire functionality is described by an interpretation function i which takes programs to their effects. Figure 3.3 shows the original PIE model.

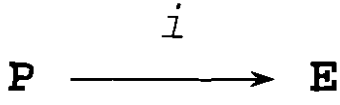

Figure 3.3: The PIE model.

The simple PIE model can be refined to distinguish between two separate features of the effect space—the *result* and the *display*. The result deals with the final end-product of the interaction between human and computer, whereas the display deals with the intermediate and ephemeral aspects of the effects. Importaut properties of the interaction can be characterized as relationships between the display—which the user usually can perceive directly and continually—and the result—which is of most importance to the user bnt is not always directly nor continually perceivable. Iu fact, one of the most common henristic guidelines for interactive system design is termed *WYSIWYG* ("What you see is what you get"), which cau be deciphered as a comparison between the instantaneous display and the end result of the interaction. Figure 3.4 depicts tbis refinement, aptly named the *red-PIE model*. We will discuss the red-PIE model in more detail in Chapter 7.

An important feature of the PIE models is the flexibility of the effect space as given by this definition from Dix [48, p. 40]:

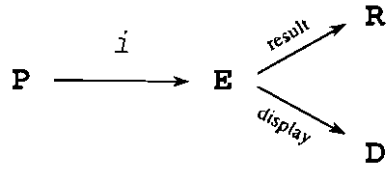

Figure 3.4: The red-PIE model.

E - the effects space. The set of all possible effects the system can have on the user. This may be thought of in different ways, and at different levels. For example, it may be regarded as the actual display seen by the user, or as the entire text of a document being edited, perhaps even the entire store of information available to the user.

The program set is also flexible. It may be regarded as the keystrokes and mouse movements, or as more abstract domain-specific operations for the system or the user. Within our interaction framework, this would mean that particular effect and program spaces lie on a continuum between the *System* and the *User*, as depicted in Figure 3.5. With such a flexible definition of the effects and programs spaces,

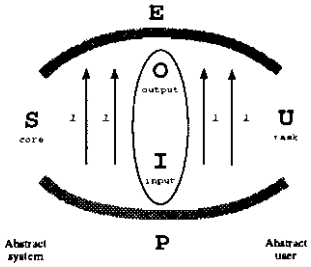

Figure 3.5: The effects space of the PIE model within the interaction framework.

it is possible to write many interactive properties and pitch them at varying levels of generality. The statement of properties within the PIE model have increasing psychological significance as both effects and programs move towards the *User*. Unfortuuately, the increased psychological significance is accompanied by a decreased design significance, as the constrncts bear less of a resemblance to constructs which can be represented in a real design.

There is a close connection betweeu the PIE model and the simple agent model defined in Section 2.2. We have defined the internal interpretation relation, I^{int}, which links sequences of stimulus events to achievable states. The sequences of stimulus events are the basic constructs of the program set. The state description of the agent is the definition of the effect set. Therefore, I^{int} is very similar to the interpretation mapping of the PIE model. The importance of this connection between the PIE model and agents is that we are able to incorporate all of the work on PIE models which express interactive properties into our work with agents. In Chapter 4, we will give examples of how some classes of abstract interactive properties first expressed in terms of the PIE model can be expressed within the agent model.

The PIE model demonstrates how a black box model can be used to formalize interactive properties. It is important to highlight some of the problems as well as the advantages of the black box model. The abstractness of a black box model has several major advantages for use in iuteractive system design and analysis. It is a very simple model that is uncluttered by the details of system implementation. This makes it possible to express properties of interaction that are implementation and domain independent. It can be represented mathematically, so the statements of properties that it allows are precise and they provide a possibility for proof. As shown by Monk and Dix [113] it can also be used as a semi-formal aid to design. An initial encroachment of valid psychological assumptions into the system modelling practice can be achieved with the black box model. This marks the beginning of a bridge between psychological theory and software engineering.

Most of the drawbacks of the black box model arise because it is so abstract. Formal techniques have been criticized for being too far removed from the practice of software engineers designing real systems. The techniques that the black box model present are not constructive, that is, they do not provide enough support for practising designers in going from an initial specification to an implementation. As far as this criticism is concerned, the black box is a move in the wrong direction. Nobody would waut to specify an entire system in terms of a PIE! And though a black box model provides the ability to precisely express properties of interaction at various levels, depending on the meaning attached to the effects space E, proof of these properties in a real specification would be a nightmare. The black box model has no way of managing the proof obligations of large specifications because it makes no attempt to modularize the description. It is in order to address these criticisms that we provide the more coustructive theory of agents. We admit that having provided a more constructive theory we are still far away from providing a compositional proof system as we would like. However, the refined agent model is a move in the right direction.

But there are more serious psychological criticisms of the approach that underlies models such as the PIE. A predictability property, which mathematically

states that the result of future input to the system is determined by the current effect preseuted to the user, claims to capture a psychologically significant feature of an interactive system at a very high level. But predictability is uot all that is important for determining the usability of an interactive system. In Chapters 4 and 7, we will provide other examples of general interactive properties relevant at varying levels of detail as well, but the question remains as to whether a 'complete' set of principles exists such that satisfaction of those principles guarantees a usable system. Our formalisms do not address this issue of completeness.

There is skepticism that the mathematically formulated properties capture the intent of their psychological cousins. This is a general criticism concerning requirements capture. Dix and Harrison [51] speak about the 'formality gap' that exists between requirement, usually couched in natural language terms, and formal specification, expressed using mathematics. The purpose of the interaction models, such as a black box model like the PIE, is to partially bridge this gap by allowing a formalism specifically geared towards psychological or HCI issues that may arise in the requirements. But the important point is that the formality gap can never be fully bridged because there will always be a translation from natural language to the formalism. This then demands that the translation be made as readily understandable as possible to the one who must verify its correctness. Since someone with psychological insight would best be suited to verify psychologically grounded requirements for usability, the interaction models must be geared to their understanding.

The black box model assumes that the relevant properties for interactive design can be expressed from first principles for any system. This assumption is reinforced by the placement of the interaction modelling component in the revised lifecycle mentioned above. There are two criticisms of this assumption. First, some believe as Carroll and Rosson [36] that design is essentially empirical 'not because we don't know enough yet, but because in a design domain we can never know enough.' Another somewhat related issue is that expressing general and abstract properties at a high level of design assumes there are features of an interactive system which can be factored out in determining the system's usability. Carroll criticizes this view by noting, along with others, that there is potentially 'infinite detail' at many levels of description in an interactive system, all of which plays a critical role in determining the usability of the system [35]. These views should not be considered a refutation of the value of a formal 'get it right the first time' approach. Rather, they should be cousidered as a warning that a formal approach alone is not sufficient for the design of usable interactive systems and it should be complemented by other techniques, such as rapid prototyping and empirical evaluation [7].

There are those who believe that an iterative approach to design with fast and easy prototyping is the only way to create usable systems. Lewis classifies these people in the "process is paramount" category of his partitioning of HCI researchers

[101]. However, initial design decisions (step 0 in the iterative approach) greatly affect the final result and the number of iterations necessary to get there. So it is advocated that some semi-formal reasoning be available to the designer in order to make good first guesses at the design based on certain general principles.[1] A black box model of interaction can allow greater confidence in initial design decisions because the properties it best supports are very general and most easily dealt with at an abstract level, which is where most design begins. And placed in a more formal design process, the assurances of step 0 design decisions can be made with greater confidence.

In summary, the importance of the PIE model is in its bridging of the 'psychological gap' in HCI. The PIE model formed the initial inspiration for the work by Sufrin and He [158] in describing interactive processes in a formal notation for subsequent analysis. This work in turn inspired the agent theory of this thesis. Barnard and Harrison's arguments motivating an interaction modelling framework suggests that there is no existing model of the interface and that all modelling techniques should fit squarely into either category of system or user model. We question that assumption, since the black box model directly discusses the model of the interface, though it does not go so far as to propose one interaction state, preferring instead to allow levels of abstraction between the system and the user, all of which represent some kind of interface. The very notion of an interaction state runs counter to the intention of the black box model because it is precisely the abstraction away from underlying state that produces the black box.

3.1.4 Software architectural models

The emphasis in this thesis is on the incorporation of psychologically valid claims about human-computer interaction into the design of interactive systems, *by means of a formal approach*. There is, however, another way to incorporate psychological principles into design, and it is by far a more common approach. That would be by a *heuristic* approach in which reasoning based on experience and empirical evidence drives the common sense applied in the design process. Figure 3.6 demonstrates the different reasoning directions one can take in applying psychological knowledge of HCI issues to the design of interactive systems.

We assume that both formal and informal advocates have access to the same body of psychological knowledge about human-computer interaction. This knowledge embodies information which describes what is meant by effective interaction between user and system. The advocates of formal methods try to capture this information in the same mathematical language which is used to describe the system. The aim of this activity is to produce a mathematically constructed usable

[1] Lewis also makes this point, but adds that "process is paramount" people doubt the contributions that formal models can provide in attaining best first guesses.

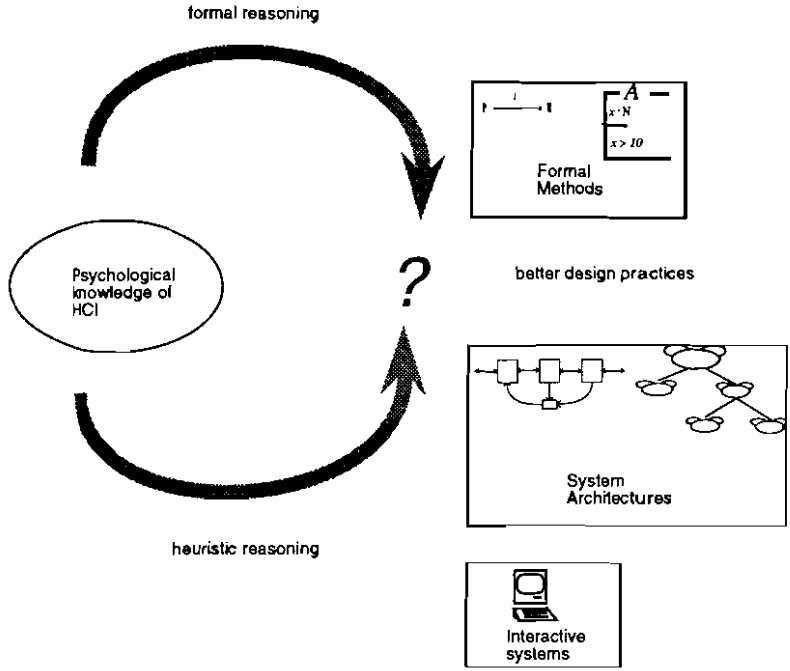

Figure 3.6: Heuristic versus formal reasoning within HCI

system which can then be refined into an executable version which preserves the usability properties. The greater majority of researchers and practitioners, take a far more pragmatic approach. They use common sense in reasoning backwards from the myriad of examples of interactive systems to arrive at abstractions from design that capture why it is that some systems are usable and others are not. In Lewis' taxonomy, these are the "systems are paramount" researchers in HCI.

Some of the abstractions provided by pragmatists we classify as interactive architectures which are themselves expressions of frameworks for interaction. We can further subdivide this class into two (somewhat overlapping) subclasses. One subclass deals with the significant contributions of the User Interface Management System, or UIMS. The other subclass deals with the more recent emergence of multiagent architectures.

An important feature of both subclasses is that they are mainly concerned with

design of the user interface. Restriction to the details of the user interface assumes the existence of the underlying application. There is an enormous economic argument in favour of this approach because it recognizes that there are many competent computerized systems in existence whose major failing is not in performing functions relevant to its domain but in providing an interface through which users can easily access that functionality. Interactive architectures have been developed with the understanding that the user interface can and should be a separate module of the interactive system. In practice, however, such a clean separation is not easy. Nor is it clear that such a separation is wholly desirable.

We contrast this with the theme of this thesis, which is interactive system development. We do not assume the existence of the underlying application in our formal analysis. On the contrary, we depend upon a system core that has been properly and formally specified in accordance with a task analysis on the domain of the application. This process ensures the possibility that user interaction, which will have to occur through an interface mediator, will be effective since the system is designed to perform the tasks that the user will require. Despite the contrast between our approach and that of the restricted user interface design, there is still good reason why we can consider it. User interface design can be viewed as a proper subset of interactive system development. If we can assume the existence of a system core which performs the tasks desired for the interactive system, then we can proceed to the description of the user interface which should preserve the task information it presents from the system to the user.

Unfortunately, the majority of existing systems do not adhere to our requirement that they respect the output of a task analysis for their domain of application. Recent work on the ESPRIT project FOCUS [58] has suggested the use of a back-end management system whose purpose it is to impose a relevant task structure to existing applications for which a new user interface is desired. In theory, the back-end manager will allow a principled design of the user interface, but there is no clear indication that their development will always be more cost effective than complete system development from the ground up.

The earliest work on user interface management systems dates back to work by Newman on the Reaction Handler [120], but it was not until many years later that the term UIMS came into use [150]. A result of the Seeheim Workshop on User Interface Management Systems was one of the first standardized architectures for UIMS, called the Seeheim model [129], shown graphically in Figure 3.7.

The main parts of the UIMS are a presentation component, a dialogue controller and an application interface. These roughly correspond to the lexical, syntactic and semantic levels of Foley and van Dam's model [59], which we call the language model. The fourth unnamed box in the diagram recognizes that for proper semantic feedback and efficiency reasons, it is sometimes necessary or desirable to circumvent the dialogue component and provide direct application interface to presentation

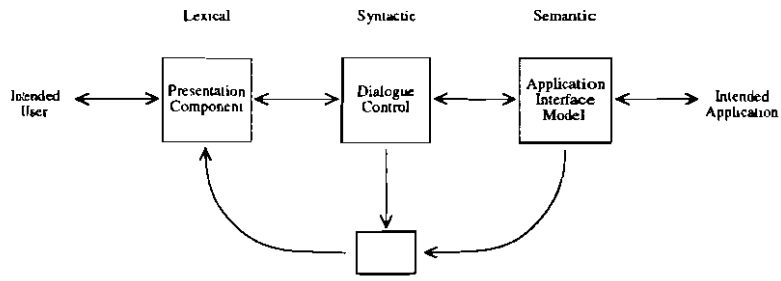

Figure 3.7: The Seeheim model of a UIMS

communication. Figure 3.8 shows how the language model fits within the interaction framework. Thus we can see the the language model gives a more complete account of the interface in our framework, but does not treat the user and system on an equal footing with the interface.

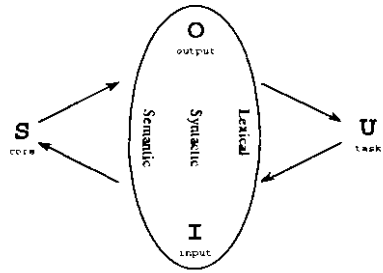

Figure 3.8: The language model of UIMS within the interaction framework

The Seeheim model has recently been revised [23], as shown in Figure 3.9, to include domain specific information and recognize the large number of toolkits available for the design of interactive objects. This new model—referred to as either the Arch or Slinky model—is intended to reflect the ability to capture domain information in interactive objects that lie close to the user.

One of the drawbacks to a model such as the Seeheim model is that although it does separate tasks within the user interface to enhance portability, it does not provide for a modular approach to the development of each part. The designer is given no aid to the structured development of a complex user interface by means of the composition of smaller and simpler interface modules. An additional model must be superimposed on the Seeheim model to facilitate a modular development scheme,

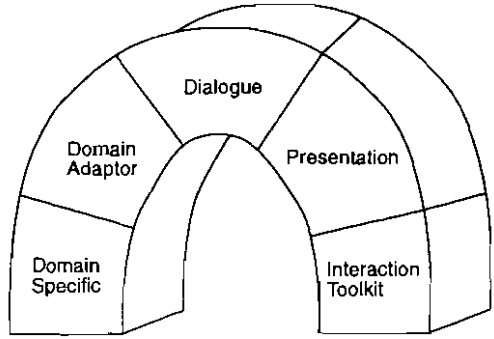

Figure 3.9: The Arch/Slinky model

and many UIMS incorporate such a model for the development of each component in the UIMS. For example, Green discusses the use of recursive or augmented transition networks, BNF notation, and event handler notations as alternative notations for development of dialogue controllers [62]. Various presentation device notations exist, such as PHIGS and GKS, with which toolkits can be constructed for handling of logical input and output mechanisms. A more general notation based on the input and output device spaces defined by Card *et al.* could replace these [31]. Also, the application interface can have its own notation, though its job as a translator from interface to application semantics strongly urges a similar model to that used to develop the application (if such a model exists!).

The apparent freedom of choice for models and notations within each component in the UIMS model suggests another problem for the designer. One of the major obligations of a user interface designer is to ensure that the interface adequately reflects the tasks of the application so that a user is free to interact with the task and not with the interface. Whereas this chore is simplified by breaking the proof into three stages representing semantic, syntactic and lexical correspondence between application and physical interface, it is not clear that the job is at all simplified by imposing very different models at each layer. It is also argued that the strict separation between layers in the Seeheim or language model does not support the handling of semantic feedback [149]. It is for these reasons that some more recent UIMS have utilized a single model in which the various components of the Seeheim model or language model can be expressed. One such example is GWUIMS [150], which is viewed as a UIMS based on the object-oriented model rather than an object-oriented implementation of another UIMS model.

Models such as GWUIMS mark the move in our discussion toward the second subclass of interactive architectures, namely multiagent architectures. Multiagent

architectures recognize that the key to building complex interactive systems is by use of a compositional model in which smaller, simpler modules can be created and composed to form larger, more complex modules. We will only mention some of those in this thesis, namely GWUIMS, the MVC model for Smalltalk [61, 29, 96] and Coutaz's own PAC model [39, 38, 40]. A somewhat related architecture has been presented by Took, in which the interaction medium—the surface—is treated as an independent agent with which both user and application interact [163, 162]. Like the language models, these multiagent architectures recognize a separation between presentation and application, but there is no uniform argument in the multiagent architecture literature as to how this separation should be done.

What separates these compositional multiagent models from the more general compositional process models of software engineering is their emphasis on the special considerations of interaction in the intended design. One of the purposes of the general agent model in this thesis is to demonstrate how a general and more formal compositional model can be used to describe interactive architectures. Our conclusion is that heuristic and formal approaches to design are complementary and a connection between these formerly distinct approaches is the key to an effective design practice. This theme is continued in Chapter 8, in which we investigate the connection between the formal agent model and multiagent architectures.

3.2 A survey of research within the framework

We will now use the interaction framework to provide a structured review of the previous formal and informal research relevant to interactive system design. Section 3.2.1 concentrates on work that has addressed the *articulation* translation from *User* to *Input*. In Section 3.2.2 we will outline the work done on the *Output* to *User* translation (*observation*). Section 3.2.3 discusses work pertaining to the translation from *Input* to *System* (*performance*) and from *System* to *Output* (*presentation*). We combine the work on these two translations because most work on one of them has been also applied to the other.

Much of the material in this section has been culled from state-of-the-art reviews by Abowd *et al.*[6] and Abowd, Dix and Harrison [1] and in a tutorial on formal methods in HCI by Harrison and Abowd [69]. Other good reviews on analytical models have been produced by Reisner [134] and Butler *et al.* [30].

3.2.1 Research on *articulation*

The formalisms which address the translation from *User* to *Input* have been developed by either psychologists or computer scientists whose main interest is in understanding user behaviour. Therefore, the emphasis in this work has been on formulating some model of the user—a *user model*. By user model in this context

we mean the designer's model of the user (see [122]). The user model is then analyzed to describe features of the interaction. This may involve either an additional description of the input (as in Kieras and Polson's CCT) or simply be done by using the input description as the user model (as is done in Reisner's use of the BNF notation). A useful summary of these formalisms may be found in the paper by Green, Schiele and Payne [63] who classify them in respect to how well they describe the *competence* and *performance* of the user. A *task* or *goal* is basic to both approaches. In practice all the notations that deal with competence and performance incorporate aspects of both to a greater or lesser degree. Quoting from Simon [151]:

> Competence models tend to be ones that can predict legal behaviour sequences but generally do this without reference to whether they could actually be executed by users. In contrast, performance models not only describe what the necessary behaviour sequences are but usually describe both what the user needs to know and how this is employed in actual task execution.

Simon goes on to classify these notations (and cognitive models in general) in a 3-dimensional space, representing various trade-offs made by their designers. We choose a simpler classification here, partitioning the formalism into two main categories:

- hierarchical representations of the user's task and goal structure; and
- descriptions of the dialogue as a language, or formal grammar.

Goals, Operations, Methods and Selection (GOMS)

The best example of the hierarchical representations is the Goals, Operations, Methods, and Selection notation (*GOMS*), based on the Model Human Processor and Keystroke-Level model presented by Card, Moran and Newell [32]. This assumes the user has a hierarchical decomposition of goals into subgoals. The goal decomposition may be deterministic or may involve choice among different strategies for achieving the goal. At the leaves of the resulting goal tree are the basic operations that the user carries out to achieve the lowest level subgoals. Granularity of the resulting analysis is given by these basic operations. So for example at a coarse level `edit-document` may be regarded as basic, whereas for finer grained analysis `press-cursor-up-key` may be terminal. Analysis of the goal structure provides measures for determining user performance. For example, the stacking depth of subgoals can indicate short term memory requirements. The models of the users' mental processes implied by GOMS are very idealized and do not apply to error conditions that will arise in the attempt to satisfy goals.

Formal grammars

Representative of the linguistic approach is Reisner's use of *BNF* notation to describe the dialogue between user and system as a formal grammar [133, 135, 134]. This views the dialogue at a purely syntactic level, ignoring the semantics of the language. Typically, grammar rules ignore computer output and the emphasis for analysis is on the complexity of the input. There are well known techniques for analysing the complexity of grammars, and these can be used to give a crude measure of the difficulty of a dialogue, however the interpretation of such measures is severely complicated by such things as familiarity with (perhaps complex) grammatical forms, clear mode changes etc. Others have used state transitions diagrams, most notably Jacob [89, 88], and added actions to grammar rules, which include output among the grammar's terminals.

Task Action Grammar (TAG)

Payne and Green have developed a notation called Task Action Grammar (*TAG*) [127, 128, 142] to describe the *consistency* of the input language by describing the user's task structure and the action sequences that accomplish those tasks. The task structure description conforms with the hierarchical goal structure of GOMS. The central role of the task in this formalism is at the expense of attention to the user. The view of the user is just as the goal structure; no consideration is given to how that structure may be modified by the results of previous interactions as observed in the output.

TAG uses parameterised grammar rules to emphasise consistency and world knowledge (e.g., up is the opposite of down). For example, assume the following is an adequate description of the user's knowledge of how to draw a graphic object, such as a rectangle, in Apple MacDraw, as shown below.

select rectangle tool	then
place mouse at one corner of the desired rectangle	then
depress button	then
drag to opposite corner	then
release button.	

This is represented in the TAG notation by means of task production rules. The use of *semantic features* such as *objecttype* allows a more general description of the task by use of user-relevant attributes, similar to the psychological attributes discussed in Chapter 2. The following is a TAG schema rule description of the task for drawing a rectangle or square (a square is a constrained rectangle).

$$task[effect = add, objecttype = rectangle,$$
$$\quad constraint = any, selecttool = any] :=$$
$$\quad\quad selecttool[objecttype = rectangle] + draw[constraint]$$

draw[*constraint* = *yes*] :=
press SHIFT + *place mouse* ...
draw[*constraint* = *no*] :=
place mouse ...

Within this claimed mental representation of the system it then becomes possible to analyse notions of consistency. Since a square is just a constrained form of a rectangle, its creation should be consistent, or similar to, the creation of a rectangle. This is captured above in the first task production rule by means of the *any* value for the semantic feature *constraint*. Heuce, a desired consistency is captured by the generic task description. Here consistency is related to the user's understanding. Conseqnently, there are clear design implications. What TAG does not attempt is to provide any linkage between appropriate task action grammars and possible designs.

Recent developments of the notation (by Howes and Payne [83]) include attempts to make good some of the limitations of TAG, in particular to include display information and flow information (no state is implied by TAG). A pessimistic view of these developments is that they make an already cumbersome notation worse. The possibility of developing informative notions of consistency becomes even more remote. Although this notion is purported to be a competence formalism [63], it is clear that the breakdown of the task into action also has performance implications. Attempts to scale TAG to substantial applications have not been satisfactory (see for example Schiele and Green [142]).

Despite these criticisms, the methodology of TAG is extremely important when considering the link between psychological and computational HCI research. The semantic features that are the basis for TAG schemas are based on empirical evidence of how users perceive the tasks they perform and the interfaces with which they interact. We will take this identification of task information from the user's perspective and use it as the fonndation for a user-centred design and analysis method described in Chapters 7 and 8.

Command Language Grammar (CLG)

Taking the lead from linguistic theory and parsing, it is often suggested that several levels of grammars ought to be used. Moran's Command Language Grammar (*CLG*) [114] is probably the most well known example of this. CLG uses four—lexical, syntactic, semantic and task. It is more design oriented than most other similar approaches, the task level being described first, obtained presumably from a task analysis, then the semantic level, formalising the entities, before moving on to the more concrete levels. Various rules are given for checking consistency within and between the levels, although these are rather loose and incomplete. This approach comes closer to viewing the entire system as involved in the interaction

rather than just the surface dialogue. Unfortunately, it has been found unwieldy to use in practice [148, 147], and the uotation used is particularly arcane, looking very much like LISP.

Cognitive Complexity Theory (CCT)

Cognilive Complexity Theory (*CCT*), first presented by Kieras and Polson [93]) combines the goal hierarchy and dialogue grammar approaches. It has two parallel descriptions. User goals are expressed as *productian rules* à la GOMS. The system grammar is given by a generalized transition network (*GTN*). The production rnles are a sequence of rules of the form

if *condition* **then** *action*

where *condition* is a statement about the contents of working memory. If the condition is true then the production is said to have fired. An *action* may consist of one or more elementary actions. The *user program* is written in a LISP-like language and generates actions at the keystroke level that have associated performance characteristics. This user program can be execnted and assessed empirically and analytically. In addition, mismatches between it and the system grammar can be found and a dissonance measure produced. The GTNs which describe this system grammar are in the form of diagrams representing the dialogne states with arcs representing the possible transitions on user actions. The difference from simple state transition diagrams is that the nodes may be hierarchically decomposed. This system part of CCT conld be executed in the same way as a grammar to give a crnde dialogue prototype.

Problems with goal structures aud grammars

The formation of the goal hierarchy is largely a post-hoc technique and rnns a very real risk of being defined hy the dialogne rather than the user. Knowles [94] attempts to rectify this by producing a goal structure based on a pre-existing manual procednre. She thus hopes to obtain a natural hierarchy. In addition, she criticizes the mechanical measures of complexity because they do not take into account issues snch as user knowledge.

In addition, as more display-oriented systems encourage less structured methods for goal achievement. Instead of well defined plans, the user is seen as performing a more exploratory task, recognizing fruitful directions and backing out of others. Typically, even when this exploratory style is nsed at one level we can see within it and around it more goal-oriented methods. So for example, we might consider the high level goal strneture below.

```
WRITE_LETTER   ==>   FIND_SIMILAR_LETTER + COPY_IT + EDIT_COPY
```

However, the task of finding a similar letter would be exploratory, searching through folders and recognizing possible places may not be well represented as a goal structure at all. Similarly, the actual editing would depend very much on non-planned activities. If we drop to a lower level again, goal hierarchies become more applicable. For instance, during the editing stage we might have the following sub-dialogue for deleting a word.

$$\begin{aligned} \textit{delete.word} &\Rightarrow \textit{select.word} + \textit{click.on.delete} \\ \textit{select\_word} &\Rightarrow \textit{move.mouse.to.word.start} + \textit{depress.mouse.button} \\ &\quad + \textit{move.mouse.to.word.end} + \textit{release.mouse.button} \\ \textit{click.on.delete} &\Rightarrow \textit{move.mouse.to.delete.icon} + \textit{click.mouse.button} \end{aligned}$$

Thus goal hierarchies can partially cope with display oriented systems by appropriate choice of level, but the problems do emphasize the prescriptive nature of the cognitive models underlying them.

Grammar techniques were initially developed to examine command-based and keystroke dialogues. One problem in applying them to mouse driven window systems is determining the lowest level lexical structure. Pressing a cursor key is a reasonable lexeme, but moving a mouse one pixel is less sensible. In addition, pointer dialogues are more dependent on the display. Clicking a cursor at a particular point on the screen has a meaning dependent on the current screen contents. This problem can be partially resolved by regarding operations such as `select-region-of-text` or `click-on-quit-button` as the terminals of the grammar. If this approach is taken, the detailed mouse movements and parsing of mouse events in the context of display information, such as menubars, are abstracted away. This means that any prototyping of the dialogue will be at a similarly abstract level or require "Wizard of Oz" techniques to mock up the full interface.

Programmable User Model (PUM)

Goal structures such as those provided by GOMS, TAG, CCT and CLG form rudimentary user models, none of which are very good at handling user error. More recent research by Young *et al.* [173, 175] has investigated the possibilities of programmable user models (*PUMs*) which can more directly address the question of error in order to further aid design of the interface. This research involves executing programs in the SOAR cognitive architecture [98] to perform *scenarios*—typical examples of user interaction with the machine—to determine usability consequences of a given system design for accomplishing given tasks. An advantage of this approach over the others is that a detailed description of user procedures is not necessary. Rather, a *knowledge analysis*, embodying the user's understanding of the functionality of the system from its intended interface, characterizes the possible behaviours of the user. The executable cognitive architecture then uses minimal problem solv-

ing techniques, familiar to the AI community, to highlight usability consequences and possible behavionral errors which would result from a means-end analysis.

The PUM methodology relies on levels of description, much like CCT and CLG. Two levels of interest are the task level and the device level. The task level description is device independent, and problem solving in that space is usually straightforward. Solutions determined at the task level are then mapped into the device level, which is given by the knowledge analysis. The PUM provides a trace of user actions at the device level and the goals that those actious satisfy. This trace of user behaviour can be compared to the designer's intended behaviour. A discrepancy would indicate to the designer a possible problem that may need correction. A simple and effective example in of this procedure in text editing is given in [173].

Graphical or diagrammatic approaches

A major criticism of formal techniques is that they are not accessible to the average designer. On the principle that many people find graphical notations easier to use, there have been many different notations proposed. Most of the hierarchical and grammar notations can be given a graphical form. In addition, there are data-flow diagrams, state transition diagrams (of many flavours), Jackson System Development (*JSD*) diagrams and simple flow diagrams. Diagrammatic notations are often used in conjunction with other notations and may have automatic support. For instauce, Marshall's diagrammatic notation [105] (see below) links Harel statecharts to VDM.

Sutcliffe [159] has nsed JSD process structure diagrams to describe tasks. He then analyzes these in order to highlight possible problems such as memory limitations (rather like GOMs). Similarly Walsh *et al.*[165] have integrated task analysis techniques with JSD. They point out that these notations are already heavily nsed for the software development side, and therefore they form a common language. JSD diagrams can be nsed quite simply as a model of the dialoguc, being a particular form of grammar.

Conclusions on *articulation*

This section has discussed the different psychological and soft computer science models of the *User* to *Input* translations. All these approaches are still at the research stage. However, the general idea that producing a description of how the user is to accomplish expected tasks in parallel to the actual system development seems useful. It is generally agreed that the form of the modelling is uot nearly as important as the discipline it enforces on the designer.

3.2.2 Research on *observation*

In this section, we overview some of the analytical methods used to assess the translation from *Output* to *User*, the *observation* translation in our framework. This is perhaps the most important translation as far as determining overall usability of an interactive system, and yet it is the most elusive. This is not surprising, since at the very core of this analysis, we are trying to determine how individuals understand that which they perceive. It is one thing to empirically test the perceptual capabilities of an average human user [112], and from that provide some model of the user as an information gatherer, but it is far more to explain how that perceived information is transformed into knowledge about the surrounding world. This topic is certainly beyond the scope of this thesis, but it is interesting to note that research which has occurred in this area. Research in this area is now generally viewed as the next major challenge for psychological research in HCI.

Most of the information on analytical models of perception has been taken from the review done at the CHI'88 Workshop on Analytical Models [30], and therefore our comments are very brief. A model not usually grouped with the analytical models is Barnard's Interacting Cognitive Subsystems (ICS) which we discuss here because of its contribution to visual processing and its apparent amenity with the agent model.

Display Analysis Program (DAP)

Tullis has produced a computer program, the *Display Analysis Program* (*DAP*), which takes as input the actual displays for a system and produces a listing of improvements that can be made in the design of the screen layout to improve the time requirement for location of specific text units on the display. His work ignores the semantics and task structure of the display. He provides support for the variance in locating textual units in terms of certain selected characteristics of the layout and perceptual attributes that can be objectively assessed by a computer.

ANets

Just as CCT and other notations mentioned above use a two layer description of goal structure and device structure to discuss the translation between the two, so does Chechile's approach to modelling comprehension of displays rely on a two level description, one of real world knowledge and one of display knowledge. Each of these descriptions are given in terms of *ANets*, augmented forms of semantic networks. The world knowledge network represents the users general knowledge about the display format and the domain of the interactive application. This would comprise knowledge about how concepts relevant in the domain (the psychological attributes in our terminology) would be represented in the display (the display attributes).

The display knowledge represents information about actual displays as snapshots in the dynamic interaction.

Cognitive Environment Simulation (CES)

Roth *et al.* have developed a symbolic processing model of the inferencing and evaluation procedures of nuclear power plant operators. The aim of the model is to provide predictions of situations aud properties of the environment and the information provided by an interactive system that will lead to errors in assessment and intention formation. CES is an example of knowledge-based simulation models of human performance whose objective is to explicitly present domain goals and the knowledge necessary to support those goals. CES is not based on a cognitive architecture, as ANets are.

Interacting Cognitive Subsystems (ICS)

Barnard attempts to incorporate two separate psychological traditions in describing his cognitive architecture of *Interacting Cognitive Subsystems* (*ICS*) [17, 18, 21]. One is the architectural and geueral-purpose information-processing approach of *Short Term Memory* (*STM*) research. The other is the computational and reprеseutational approach characteristic of psycholinguistic research and AI problem-solving literature.

ICS provides an architecture for perception, cognition and action built up by the coordinated activity of nine smaller subsystems, five comprising a peripheral subsystem in contact with the physical world and four comprising a central subsystem. Each of the uine subsystems is specialized for handling some aspect of external or iuternal processing. For exaınple, one peripheral subsystem is the visual system for describing what is seen in the world. An example of a central subsystem is one for the processing of propositional information, capturing the attributes and identities of entities and their relatiouships with each other (a particular example is that propositional informatiou represents "'knowing' that a particular word has four syllables, begins with 'P' and refers to an area in central London.")

A subsystem is described in terms of its typed inputs and outputs along with a memory store for holding typed information and transformation functions for processing the input and producing the output. So ICS can be seen to have a natural description in terms of agents, but just how close the correspondence is between the ICS subsystems and the agents presented iu this thesis is an interesting questiou which, unfortunately, remaius open.

Though ICS is purported to provide a model of perception, cognition and action, the type of information it provides on the action side is not the same as provided by the models for action described in our discussion of models for *articulation* above. ICS is not intended to produce a description of the user in terms of sequences of

actions that are performed. ICS provides a more holistic view of the user as an information processing machine. The emphasis is in determining how easy particular procedures of action sequences become as they are made more automatic within the user. The lack of quantitative ontpnt from ICS makes it less practical than the other *articulation* models. However, ICS does provide competent and understandable analyses of how visual information is perceived and transformed by the user, and this information provides substantial advice to designers wishing to enhance the proceduralization of user behaviour, since proceduralized behaviour is assumed to be less prone to error and, hence more effective.

The main purpose of ICS is not so much as a model for generating a description of a user's behaviour, as one views the purpose of a model such as PUMS. Rather, its purpose is as a classification method for the wealth of empirical psychological evidence on user behaviour. Thus, we would expect ICS to perform better as a rationalization of user behaviour based on empirical evidence, whereas PUMS we would expect to produce better predictive information.

Conclusions on *observation*

Research on the translation from *Output* to *User* is not as progressed as research on the translation from *User* to *Input*. This is perhaps due to a lack of empirically validated psychological theory concerning perception and understanding, or at least a failure to apply those theories to questions of human-compnter interaction. Contributions from this research, however sparse, will bear a great significance in directing design of interactive systems.

3.2.3 Research on *performance* and *presentation*

Research in this section focnses on features of the interaction which are directly affected by the *System* and the interface (*Input* and *Output*). We have bundled together the *performance* and *presentation* translations because most models devised to deal with one also deal with the other. The work in this section fits along a continuum from very abstract models (such as the PIE) to very constructive notations (such as Alexander's SPI). We will try to present this review from the more abstract towards the constructive.

Extensions to the PIE model

We have already presented the *PIE* and *red-PIE* models in Section 3.1. In order to study more specific areas of interactive behaviour, further refinements to the basic model have been introduced.

Handle spaces

A client-server relationship is a common way to view the independent execution of several programs (the clients) under one controlling program (the server). This view can be readily extended to situations in which a user plays the role of client or server.

An example of the user as client is seen in a multi-windowing system, where different windows represent different tasks and so the user treats them as independent interactive systems. Each window can be treated as a red-PIE, and so the multi-windowing system is represented as a collection of PIEs. The user is able to direct input to particular windows, i.e., the user can select a window for input. This selection can be modelled by giving each window as a red-PIE a *handle* for distinguishing its input. Alternatively, there is an injective function from handles to PIEs. The overall display and result functions will depend on the set of active handles, or active windows, yielding a red-PIE description of the multi-windowing system. Since we assume the user considers the separate windows as independent, interference between windows is undesirable. We can therefore formulate a constraint on the collection of window PIEs, called *result independence* by Dix, which would ensure that the contribution from interaction with one window to the overall result be separable from that of any other window. Handle spaces and result independence are dealt with in more detail by Dix and Harrison. [50, 48].

Situations in which the user is the client (and hence, there are many users) fall under the research theme of computer supported cooperative work (*CSCW*), a topic which is outside the scope of this thesis. A formal methods approach to CSCW has not yet been attempted, though we suspect that a treatment analogous to handle spaces would be enlightening.

Nondeterminism

There are several notions of nondeterminism relevant to interactive systems. Nondeterminism arises from the loss of knowledge, either deliberate or not, of some aspects of the whole system. The user, who for reasons of inexperience or a conscious decision to remain ignorant, does not know exactly what aspects of the system determined its behaviour. It can arise that different situations which visually appear the same react differently to the same commands. So, for example, in a graphical drawing package the selection algorithm will take into account a structural or temporal hierarchy (into layers, perhaps) of the objects which may not be fully manifested in the visual presentation. If two objects appear to overlap and the mouse cursor is placed in a location which is 'covered' by both objects, a subsequent mouse click to select will choose one or the other object depending on which is considered on top, or it may choose both, if the two objects have been grouped as one. The point is that there is usually no visual indication to aid the

user in predicting the outcome of select; the algorithm is nondeterministic as far as the user is concerned even though it may have a perfectly deterministic description at some level of detail.

Dix has discussed this nondeterminism [47] in interactive systems and how it can be modelled in a PIE. In Chapter 4 we extend Dix's ideas in terms of agents when we investigate this user nondeterminism with respect to predictability.

Temporal models

One feature of an interactive system which greatly affects its perceived usability is the avoidance of display lag wherein the current display does not adequately reflect the state of the system resulting from all prior user input. Dix [48, 49] has discussed display lag and whether it can be avoided in any system. The conclusion is that rather than chase the "myth of the infinitely fast" machine, HCI research can concentrate on what it means to make a system usable that will admit the inevitable delays. A simple extension to the PIE model in which a single *null* input event is introduced in the program language P leads to a definition of steady-state behaviour and the consequences on predictability of allowing buffered user input. One of the most important results of this work has been a formal representation of the requirements that can be placed on a system which experiences display lag in order to expose information that must be available to the user.

Levels of system description

As we explained in Section 3.1, a PIE analysis can be given at varying levels of abstraction. As Carroll has pointed out [35] important interactive features become apparent depending on the level of detail in the system description. Though we do not adhere to Carroll's further and fatalistic belief that there are an infinite number of such levels—each of grave importance to the overall usability of a system—and, hence all attempts to model such properties will be hopelessly incomplete, it is important to consider how the varying levels of detail can be related.

We can view the description of an interactive system as a collection of PIEs, each representing the system at different level of abstraction and each capturing relevant interactive properties along the way. The system description is completed by providing mappings between the program spaces (parsers, as Dix calls them) and effect spaces (projection or embedding mappings) of the various levels. The result space will be associated to a more abstract level PIE as the concrete level PIEs usually deal with more immediate and temporary features of the interaction (such as editing a command line or displaying a pop-up menu). Each PIE will contribute to the overall display. Figure 3.10 depicts these relationships in a two level description.

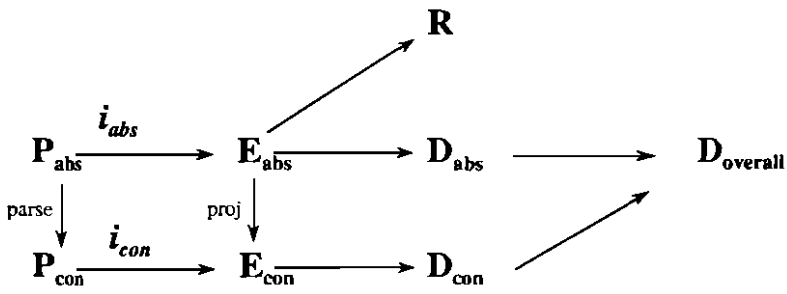

Figure 3.10: Relationships between levels of PIEs

State display model

Another obvious way to separate into levels is by considering the interface separately from the underlying system, as is suggested in the interaction framework. Harrison and Dix have considered such a state display model [73] in an attempt to formalize notions behind direct manipulation interfaces. This model considers the underlying system and the display as separate machines, similar to agents. The link between these machines is referred to as *state display conformance* in which operations performed on the display by the user are adequately mirrored by operations performed on the state. State display conformance can be used to assess the quality of a graphical interface, as Ahowd, Dix and Harrison have shown [2]. The theory of data refinement is close to that of state display conformance and we will discuss this in terms of agent correspondence in Chapter 4.

Templates

One of the problems with system models is that although they describe interactive behaviour they have no conception of how the user sees the system. We demonstrate what we mean by an example from [20] in the context of the predictability principle. One way [53, 158, 9] of making a principle of predictability precise is to require that if the effect of any two input programs are the same, then no future experimentation will betray any difference in effect between the systems. Viewing programs as sequences of inputs, we can formalize this as below.

$$\forall p, q \in P \bullet i(p) = i(q) \Rightarrow \forall r \in P \bullet i(p \frown r) = i(q \frown r)$$

This notion stresses that the *effect* is sufficient to determine the equivalence of distinct system states. From a user's point of view the fact that the two effects are

identical may not be sufficient. A stronger requirement of predictability (which as it happens may be too strong for any realistically complicated interactive system) is that the equivalence of the effects will depend on the *user's perception* of whether the effects are equivalent. If we regard the display as what the user perceives then it may be more appropriate to define predictability as below.

$$\forall p, q \in P \bullet display(i(p)) = display(i(q)) \Rightarrow \forall r \in P \bullet i(p \frown r) = i(q \frown r)$$

This says that if the viewable effects are the same, then the obtainable effects are indistinguishable (though not necessarily visibly indistinguishable). By indistinguishable we mean that no experimentation by means of input r would betray a difference.

We can go further than this and add structure to the model to incorporate claims about, say, user attention. Certain components of the display are more likely to be noticed in making decisions about the next action than others. Some parts of what is seen of the system will be different in a way that is irrelevant to the future of the application (e.g., more general system status information and the time and date).

One way to add structure to the model is through the use of *templates* as suggested by Harrison, Roast and Wright [74]. Instead of maintaiuing a single display and result function, we can allow for many display and result mappings on the effect space, each of which we will refer to as either a *display template* or a *result template*. It is beneficial to formulate properties of a design in terms of the relationship between the ephemeral and immediate display and the end result. In essence, that is precisely what the predictability property above is doing. Since we have access to a result function in a red-PIE, it would probably be beneficial to rewrite the above predictability property including the result.

$$\begin{aligned}&\forall p, q \in P \bullet display(i(p)) = display(i(q)) \Rightarrow \\ &\quad \forall r \in P \bullet result(i(p \frown r)) = result(i(q \frown r))\end{aligned}$$

This says that if the viewable effects are the same, then the obtainable end results are indistinguishable (though still not necessarily visibly indistinguishable).

The main use of templates by Harrison, Roast and Wright was in an investigation of cyclic properties of a bibliographic database. A structure that is particularly common in menu-based systems is a *cycle*. A main menu presents an initial set of options, each leading to a dialogue sequence. The sequence affects some change in the result only at its conclusion, when the main menu is redisplayed and the cycle is complete. Hence in the database, a `select` entry in the main menu will lead to a cycle that results in the selection of a reference given a particular name or date or source. Recognition of the cycle is especially useful to the designer if the user understands and relies on it for effective interaction. It then becomes important to ensure that some display template at the start and end of the cycle is clearly

significant to the nser and that whatever effect the system has on the associated result template takes place at the end. The cycle is then, in some circumstances, a mirror of the task as it would apply in the nser model. Claims about the interaction from a task point of view, perhaps the identification of possible error situations, can thcn be formulated in terms of the cycle and experimentally observed through evalnation of actual use.

Templates are determined from empirical evidence, as were the semantic features of TAG. The use of templates marks an initial encroachment of solid psychological information into the system modeling compouent of HCI. A more constructive view of templates in terms of attributes is one of the main reasons for a refinement of the agent model in Chapter 5. Templates, therefore, form the basis for an interactive design and analysis method described in Chapters 7 and 8.

Limitations and application of abstract models

The biggest danger of any formal approach is that the designer may attach more credence to it than it deserves. This is particularly true of abstract models of the sort described here. An unwary user of the techniques may believe that formal consistency with some of the principles and models described above was *sufficient* for a system to be usable. The fallacy of this is obvious to anyone with an understanding of HCI, but formalisms can be both seductive and blinding. It is therefore essential to understand the limitations of the abstract techniques above.

Rather than being snfficient conditions for usability, the various formal statements of principles tend to be *necessary* conditions. A formalization of the familiar slogan "what yon see is what you get" (*WYSIWYG*) as done by Dix [48] and Thimbleby [161] and Sufrin and He [158] says that it is possible to work out the end result of a system by interactively examining it. This is essential if the system is to be usable at all, but does not tell us how easy it is to work out the result, or how visually and spatially faithful a representation of the result we see on the screen. The latter of these problems is perhaps easy for the designer to verify, and mistakes will be obvions, hut the former requires a deep understanding of human cognition that is unlikely to be formalizable to the same extent.

Some of the psychological issues are just beyond the scope of the models, and one can simply note that even when the system has passed the formal tests, more human centred analysis must be applied. Indeed, the introduction of templates was specifically geared to marrying together the abstract principled approach with an evaluative approach. In order to satisfy the formal principle the designer would have to give the relevant template fnnctions. These can then be validated either empirically or by a hnman factors specialist. Similarly, the *strategies* by which the user caninvestigate the system state [48] are a form of nser program, and can thus be validated against executable cognitive models, or again by direct evaluation. Often

the simple fact that the operations and deductions that the user must perform have been explicitly stated as part of the formal proof will be sufficient to see whether they are reasonable or not.

The other major non-formalisable part of the use of these models is deciding which principles are applicable and desirable to a particular application, and also at what degree of abstraction to apply them. A system may be understood at various levels of abstraction, such as concrete keystrokes and mouse actions, syntactic units, or semantic commands. The models can often be applied to the system at each level. Some of the properties will be universally applicable to all systems at all levels, but in general the designer will be more selective. So for instance, in a command based operating system, one expects to have a total view of the current command being edited, that is, it obeys a very strong visibility principle. However, when the command is submitted (entering the carriage return key) the results of it on the file system will be far less visible, usually requiring explicit commands to view files, directories etc. Arguably, in this example, the semantic level could do with being more visible too, but it is a design decision as to what degree of visibility is required at which level.

In short, as with any method or model the domain of applicability of abstract models must be born in mind when they are used or evaluated.

A further and related issue involves the refinement of one abstract level in order to more closely approximate an executable system while still preserving the interactive properties. Having produced specifications that satisfy desirable properties, there is no guarantee that the structures used in the specification designed to match the user model will be well-suited to implementation

The major aim of the above models has been to *define* useful properties. There is then the issue of actually building systems that satisfy them.

Having produced specifications that satisfy desirable properties, it is probable that the specification structures designed to match the user model are very ill-suited to implementation. This conflict will arise with *any* specification of interactive systems. If a designer leans towards efficient implementation structures, then it is likely that user requirements are badly defined, whereas if she leans towards the user then inefficient structures result. Hence the desire in this thesis to provide a more constructive approach via agents that is at once amenable to the kind of analysis of the abstract models but also sympathetic to software engineering.

But our agents are certainly not the first constructive approach to the application of formal methods in HCI, and we will review those now.

Dialogue specification

A problem with the use of general specification techniques is that they are too general. The dialogue of the interaction will not be separate in the notation, instead

it must be modelled. The dialogue component may be packaged and used within the standard notation, allowing the free mixture of interfacc and standard forms. This gives maximum expressiveness, but at the cost of losing the dialogue/application separation which is freqnently seen as desirable. In particular it makes it hard to analyse the dialogue structure as a separate part of the system.

There have beeu various attempts to add dialogue specification components to standard notations. These may be simply sugarings that are then translated into the underlying notation to give them semantics, or have a separate level of semantics given them. In either case, the actual concrete notation makes a clear separation between the two styles of specification.

EPROL

Hekmatpour and Ince [79], for instance, have a separate user interface design component in their *wide-speetrum langyage EPROL*. This interface component seems rather disappointing however, being simply a teletype forms and menu description snch as may be included with many data-base languages or fourth generation languages. The dialogue is apparently described entirely within the main specification language and may thns be easily obscured.

Statecharts and VDM

Marshall [105] has merged a graphical dialogue specification technique based on Harel's statecharts [67, 68] with *VDM*. This includes standard constructs such as sequence, choice and iteration in the dialogue, each terminal dialogue "box" is related to a piece of VDM specification. She also suggests that the user's actions in this can be represented by a parallel diagrammatic/VDM description, but in her examples this diagram consists of a single hox, so the claim is not supported. This exposes the fact that the diagrammatic notation does not support parallel activities (such as multi-window dialogues). It would he quite easy to add such a construct at the diagrammatic level, but the meaning when translated into VDM semantics would not be clear. The actual acceptance of input is handled by "shared" global variables with the user "process" and is hardly clean. Another problem, is that each piece of VDM works on global variables, making it difficult to trace the semantic impact of particular user actions without analysing the pieces of VDM in detail, which runs counter to the desires of dialogue separation.

A graphical notation has the considerable advantage that it is easier to comprehend initially. A disadvantage of a graphical notation is that it is difficult to formulate properties, or theorems, within a strictly graphical notation. Since our work emphasizes the use of a formal notation for the precise expression of interactive properties that can then be used to analyze and develop an interactive system, this latter disadvantage far outweighs the advantage of comprehensibility. It is a

reasonable objective in the future to extend our agent model to include a graphical representation.

SPI

Alexander [10, 11, 12] has designed an executable specification/prototyping language around *CSP* and *me-too* called *SPI*, (specifying and prototyping interactiou). Me-too is an executable specification language based on VDM and implemented under several dialects of LISP. The CSP forms the dialogue specification part, whereas the me-too supplies the semantics. This is rather similar to the way *CCS* and ACT-ONE are combined in *LOTOS*[87, 164].

SPI's dialogue component is called *eventCSP*, it includes most CSP constructs, sequential action, choice, iteration, and most importantly parallel composition. The parallel composition makes it possible to express concepts such as the choice between mouse and keyboard input. The expression of choice is based on the occurrence of events and is thus more clear. It inherits drawbacks from CSP however, such as the lack of direction in events, it is not evident iu the syntax whether an event is due to external input, produces external output or is an internal synchronisation between parallel processes. This can be confusing in dialogue design when there is an obvious direction of control flow. However, the problem is largely mitigated for user I/O by the judicious choice of event names. It is thus only internal events that remain confusing. The structure of possible events is static too. This would make it hard to deal with the dynamic creation of windows for instance. This lack of dynamic configuration (and related lack of parameterisation) is common to many dialogue languages, it would be easy to add to most, but would typically reduce the possibility of analysis of the dialogue structure.

The semantic part of SPI is called *eventISL*. Although it is based upon me-too, it is intended to operate with various languages, in particular a *C* version is available. The host language independent part consists of several elements: a clause giving the global values needed for the event, a pre-condition expressing when the event can occur, output and input parts. The host language part, simply describes what updates to global values are possible. The globals used and updated are made explicit and thus tracing the effects of events is easier. It would be possible to use other specification notations such as an algebraic notation or Z as the host language, with a subsequent sacrifice in executability, a choice we consciously make (at least initially) in designing the agent language.

SPI has a prototyping tool for use when only the eventCSP dialogue description has been produced. This allows the designer to examiue possible event traces. Later full prototypes using the me-too version of eventISL or the C version can be executed. One drawback with the implementations is that they do not offer the full parallelism of the CSP. This is because the underlying languages they

were built upon did not allow full multiplexed, non-blocking I/O. They fake the nondeterminism as long as they can for internal events, but when one of several choices of user input device are possible, the system makes an arbitrary choice. Most versions of C on UNIX or PCs have system calls for non-blocking I/O, so it should be possible to rectify this, at the cost of some loss of portability.

Anderson's work

Anderson has built on the claims of the PIE model with a more constructive refinement of it[14, 15]. He provides explictly an input language generated by a context-free grammar, whereas the input language of the PIE model imposes no such structure on the dialogue. A state description is separated from the ontpnt, with a meaning function mapping input constructs to state transitions and a display function mapping the resulting state to its output presentation. As such, Anderson's model contains three of the components found in our own interaction framework, though we represent both *Input* and *Output* as state machines and he does not. The important contribution of Anderson's work is that he provides concrete examples of a system, a file browser, and translates the abstract properties expressed as theorems on the specification and then proves their satisfaction.

Interactive processes in Z

The work of this thesis is based on another constructive refinement of the PIE model given by Sufrin and He's description of the design, analysis and refinement of iuteractive processes [158]. He [76, 77] and Josephs [92] independently gave descriptious of a state-based version of CSP in which the events of the language were coupled to transitions of an underlying state machine. Sufrin and He provided the description of the state machine in the Z language and encapsulated a CSP-like syntax within Z for the event specification. The importance of this work can be seen in several ways.

The PIE model served as the motivation for the interactive processes. One of the requirements for interactive processes was that they would be able to express the same types of properties expressible with a PIE. Sufrin and He convincingly satisfied this requirement, ultimately providing a very systematic and intriguing categorization of WYSIWYG-like properties in terms of the display and resnlt mappings. Our own derivations of interactive properties iu Chapters 4 and 7 are based on this technique. The link between the CSP and Z is similar to one suggested by SPI and mentioned earlier, but this particular work showed more of the advantage of such an approach by showing its use in formulating properties on tbe specification that could then be the subject of proof and refinement.

Z is becoming increasingly familiar in both academic and industrial circles, so the presentation of interactive processes in Z showed convincingly that HCI

concerns could be expressed in the language of software engineering, not so much as a separate and disparate module in development but as an integral component with the software development lifecycle.

Conclusions on *performance* and *presentation*

An immediate conclusion arising from this review of system based work on formal methods in HCI is that except for the Sufrin and He model, there has been little effort in the last four years on incorporating the abstract principles motivated by the PIE model into a more concrete design practice. One of the principal aims of this thesis, therefore, is to investigate a further constructive formal method and its links to interactive system design and analysis.

He and Josephs' work on state-based CSP strongly suggests that the monolithic presentation of interactive processes can aud should be made as compositional as a process algebra allows. Outside of interactive system design, there has been an increasing interest in the development of model-oriented specification techniques which more adequately address the modularization necessary for the description of large systems. In Chapter 6 we will discuss several of the attempts to extend the Z notation to capture object-oriented techniques. The agent language developed in this thesis is an attempt to address both the need for better modular specification notations and also provide a flexible and consistent notation to express internal state-based behaviour as well as external event-based behaviour. The resulting notation will allow us to formally express interactive properties at the task level. This implies that the agent model can be used in an interactive design method which is linked to task analysis information.

3.3 Conclusions

We have seen how the interaction framework of Chapter 2 was influenced by pre-existing general accounts of the structure of interactive systems. The contribution of the framework is two-fold. It first provides a uniform language for describing the three major components of an interactive system, the *User*, the *System* and the mediating interface (in terms of *Input* and *Output*). The unified treatment of these three components opens up the possibility for a cross-fertilization of research on both the system side and user side of interactive system analysis and design.

The second contribution of the framework is that it provides context for research in HCI. Having completed this contextualization we can see where the work in this thesis fits in the general scheme of interactive system analysis and design. We intend to provide a constructive model of the *System*, *Input* and *Output* in terms of the agent model and its associated language. This model will provide the means for a compositional description of realistic interactive systems. We will concentrate in

our descriptions on revealing those features of the design which highlight properties of the interaction between the user and the system. In doing so, we will provide formal accounts of existing interactive design heuristics which can lead to a more principled design practice.

In the next chapter we will describe both formally and informally interactive properties that are derived from the interaction framework. In order to relate these properties to a design and analysis methodology, we will need to provide greater detail on the structure of agents in the refined model of Chapter 5. Having refined the agent model, we will provide a design language for agents in Chapter 6 which matches more closely than the standard Z notation how a designer thinks of describing an agent. In Chapters 7 and 8 we will link the refined agent model and notation to a design methodology for generating descriptions of interactive systems and analyzing existing system based on empirical evidence of how user's understand the tasks the system snpports.

Chapter 4

Properties of interactive systems: Part I

The purpose of this thesis is to provide a means for designing interactive systems which can be analyzed with respect to desirable properties to enhance usability. We have established the context of this research in the previous two chapters. In this chapter we will investigate the kinds of properties which influence usability.

We present a catalogue of interactive properties, discussing each entry in terms of the interaction framework and/or the simple agent model. This catalogue is not intended to be complete in the sense that it lists every property of an interactive system that could possibly affect usability. The artillery with which we are equipped at this point is not powerful enough and it is even doubtful that such a complete catalogue could be compiled. We first inspect the interaction framework to uncover properties of the translations which affect overall usability, providing examples within real systems. The properties of interest for translations in the framework concern the ease associated to the translation and the coverage of the translation, and we can attempt to formalize those notions. Other interactive properties have been discussed in the PIE model literature and the Sufrin and He model which pertain to the relationship between the *System* and the *Input* and *Output* components of the framework. Assuming a compositional model of the agent, which we will define explicitly in Chapter 5, these properties can be defined over a single agent.

Overview of chapter

In Section 4.1 we will give an informal account of properties of translations within the interaction framework. The intent of this informal section is to motivate how the coverage and ease of a translation affects usability. In Section 4.2, we begin the formal treatment with the definition of a definedness ordering on translations. Though this ordering directly addresses coverage of a translation, we see its use

mainly as the basis for a refinement ordering on the external behaviour of agents. We do not pursue refinement of agents any further in this thesis. In Section 4.3, we address the ease of a translation as a correspondence relation between agents similar to treatments of data refinement. The remaining sections of this chapter apply the Sufrin and He method for classifying interactive properties as relationships between input history and state or response history. We conclude the chapter by indicating how the agent model must be altered to accommodate more salient descriptions of interactive properties relating results and displays.

4.1 Properties of translations

Figure 2.3 depicts four separate translations between the components of the interaction framework. Properties of those translations have consequences toward the overall usability of the interactive system. In this section we will discuss qualitatively how those translations affect usability before attempting to formalize properties of the translations in the next section.

4.1.1 Hutchins, Hollan and Norman distances

Hutchins, Hollan and Norman have used the execution/evaluation cycle of interaction as the basis for an informal understanding of direct manipulation [85]. Crucial to their qualitative account of the ease of use that results from direct manipulation interfaces are two notions of distance—*semantic* and *articulatory*—both of which are minimized in effective interactive systems. They classify input and output languages in the more general category of interface languages. Any expression within an interface language has both a meaning and a form. Semantic distance concerns the translation between the user's intentions and the meaning of the interface language. It is measured by the expressiveness (Is it possible to say what one wants to say in this language?) and conciseness (Are simple task expressions translated into complicated input expressions?) of the input language. The articulatory distance is a measure of the correspondence between the meaning and the form. This can be measured in terms of the link between the structure of the input or output terms and their intended meaning. For example, input to a graphical drawing package will be assisted by gesture input with a mouse, or touch-sensitive screens will be used to provide more natural indicative input (e.g., move *that* item to *there*). Monitoring a variable in a process control system may be aided by output in the form of a meter reading or a continuous graph, instead of a semantically equivalent table of time-stamped values.

The translation from user intention to user input is influenced by both the semantic and articulatory distances associated with the input interface language, as the user translates from the goal in the task language to the meaning of that goal

and its subsequent form within the input interface language. Similarly, semantic and articulatory distances affect the translation from user-perceived system output to user assessment. Semantic and articulatory distance measures are applied to the execution as well as the evaluation phase of interaction. They attempt to answer questions about the possibilities enabled by the translation and the ease of the translation the user and the interface. Figure 4.1 portrays the Hutchins, Hollan and Norman sense of articulatory and semantic distances within our interaction framework.

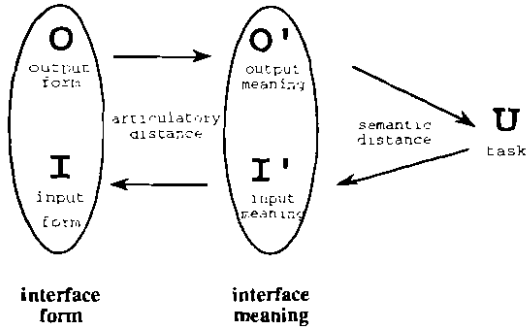

Figure 4.1: Hutchins, Hollan and Norman's distance concepts within the interaction framework

Had Norman's initial execution/evaluation cycle been more detailed on the system side, it may have prompted Hutchins, Hollan and Norman to have extended the definition of articulatory and semantic distance to the correspondence between the system and the input and output languages of the interface.

Since Hutchins, Hollan and Norman have not provided a quantifiable metric to these notions of articulatory and semantic distance, we feel uncomfortable referring to them as distances. It is for this reason that we no longer refer to the descriptions of properties on translations as distances. [1]

4.1.2 Articulation

The *User*'s formulation of the desired task to achieve some goal needs to be articulated in the input language. The tasks are responses of the *User* and they need to be translated to stimuli for the *Input*. As pointed out above, this articulation

[1] In an earlier paper [5], qualities attached to each translation in the framework were referred to as distances

is judged in terms of the coverage from tasks to input and the relative ease with which the translation can be accomplished. The task is phrased in terms of certain psychological attributes that highlight the important features of the domain for the *User*. If these psychological attributes map clearly onto the input language, then articulation of the task will be made much simpler. An example of a poor mapping that is common in our everyday lives occurs in a large room with overhead lighting controlled by a bank of switches[2]. Many times it is desirable to control the lighting such that only a certain section of the room is lit. We are then faced with the puzzle of determining which switches control which lights. The consternation resulting from repeated experimentations with the switches to achieve the desired lighting effect can be traced to the difficulty of articulating a goal such as "Turn on the lights in the front of the room" in the input lauguage, which consists of a linear row of switches which may or may not be oriented in a manner suggestive of their operation.

Examples in which articulation affects the ease or possibility of interaction with a computer systems are apparent as well. Much of the allure of virtual reality systems is achieved by novel input devices, such as a data glove, which are specifically geared toward easing articulation. Speech recognizers make it possible for disabled users to input text when typing via a standard keyboard is impossible. Systems which track the eyes of the user can make pointing tasks possible when input through a mouse or touch are not possible.

4.1.3 Performance

At the next stage, the responses of the *Input* are translated to stimuli for the *System*. Of interest in assessing this translation is whether the translated input language can reach as many states of the *System* as is possible using the *System* stimuli directly. This is similar to the notions of reachability defined by Dix and Runciman [54, 48] in the PIE model. For example, the remote control units for some compact disc players do not allow the user to turn the power off on the player unit, hence the off state of the player cannot be reached using the remote control's input language. On the panel of the compact disc player, however, there is usually a button which controls the power. The ease with which this translation from *Input* to *System* takes place is of less importance because the effort is not expended by the user. However, there can be a real effort expended by the designer and programmer. In this case, the ease of the translation is viewed in terms of the cost of implementation.

[2]This example comes courtesy of Donald Norman [125]

4.1.4 Presentation

Once a state transition has occurred within the *System*, the execution phase of the interaction is complete and the evaluation phase begins. The new state of the *System* must be communicated to the *User*, and this begins by translating the *System* responses to the transition into stimuli for the *Output* component. This presentation translation must preserve the relevant system attributes from the domain in the limited expressiveness of the ontput devices. The ability to capture the domain concepts of the *System* within the *Output* is a question of possibility for this translation.

For example, while writing a paper with some word processing package, it is necessary at times to see both the immediate surrounding text where one is currently composing, say the current paragraph, and a wider context within the whole paper that cannot be easily displayed on one screen (say the current chapter). When moving files between the directories in a hierarchically arranged file system, such as Unix[3], it is necessary for the user to know the current position in the directory structure in order to use relative path names effectively.

The Information Visnalizer developed at Xerox PARC uses three dimensional presentation techniqnes to provide more context in displaying hierarchical information that would not otherwise be possible in two dimensions [33, 138]. They give the example of the display of the hierarchy of job positions in a large company. A standard two dimensional display of such an organizational chart would be far too large to display on a display screen. A three dimensional representation in the form of a cone-tree allows the whole chart to appear on the display with a readable portion on top. The advantage of the cone-tree is that it more easily displays the context within which a portion of the hierarchy is viewed. At all times it is possible to trace the path along the hierarchy from a viewed portion to the root.

The importance of this translation is that differences between two *System* states be present in the *Output*. Hidden differences result in an increased burden on the user who is trying to assess the result of previous input relative to a specific task goal while also trying to predict the outcome of future interactions based on the current output. Lack of ambignity between distinct *System* states and the *Output* is at least a necessary condition for overall predictability and proper goal assessment.

4.1.5 Observation

Ultimately, the user must interpret the output to evalnate what has happened. The response from the *Output* is translated to stimuli for the *User* which trigger assessment. The observation translation will address the ease and coverage of this final translation. For example, it is difficult to tell the time accurately on an

[3]Unix is a registered trademark of AT&T Laboratories.

unmarked analogue clock, especially if it is not oriented properly. It is difficult in a tty interface to determine the result of copying and moving files in a hierarchical file system. Typesetting a report using one of the popular typesetting programs available today is made virtually impossible without some previewing facility which allows rapid (and tree-saving) feedback to assess progress.

4.1.6 Assessing overall interaction

The interaction framework is presented as a means to judge the overall usability of an entire interactive system. In reality, all of the analysis that is suggested by the framework is dependent on the current task (or set of tasks) in which the *User* is engaged. This is not surprising since it is only in attempting to perform a particular task within some domain that we are able to determine if the tools we use are adequate. In my own experience, I have found that different text editors are better at different things. For a particular editing task, I choose the text editor that I believe is best suited for interaction relative to the task. The best editor, if I am forced to choose only one, is the one that best suits the tasks that I most frequently need to do. Therefore, it is not too disappointing that we cannot extend the interaction analysis beyond the scope of a particular task.

A simple example of programming a VCR from a remote control shows that all four translations in the interaction cycle can affect the overall interaction. Ineffective interaction is indicated by the user not being sure the VCR is set to record properly. This could be because the user has pressed the keys on the remote control unit in the wrong order; this can be classified as an articulatory problem. Or maybe the VCR is able to record on any channel but the remote control lacks the ability to select channels, indicating a coverage problem for the performance translation. It may be the case that the VCR display panel does not indicate that the program has been set, a presentation problem. Or maybe the user does not interpret the indication properly, an observational error. Any one or more of these deficiencies would give rise to ineffective interaction.

Throughout this section, we have been referring to the two features of translations, the ease with which they happen and the possibilities or coverage they provide. With overall interaction, we can make similar distinctions between the ease of assessing the result of the previous interaction with the intended goal and the ability to achieve that goal. It is desirable to make assessment as easy as possible. However, if attainment is difficult at the same time, then the user is forced to traverse the interactive cycle many times, and this is not desirable. The best system would maximize ease of assessment and goal coverage.

More examples of how the interaction framework can be used to assess the overall effectiveness of interaction have been provided by Abowd and Beale [4, 5]. We will move on now to show how coverage and ease of translations can be formalized.

4.2 Formal properties of translations

A translation takes sequences of one language of events—the source language—to sequences in another language of events—the target language. In the interaction framework, the translations of *articulation*, *performance*, *presentation*, and *observation* each have their source and target languages fixed by some component. We introduce the generic set of translations with fixed source and target languages.

$$Translation[S, T] == S \leftrightarrow T$$

We refer to elements in the domain of a translation as source elements and elements in its range as target elements. We can define properties of the translation relation in terms of its *possibility*, *coverage* and *ambiguity*.

Possibility indicates how much of the source language is translatable. A translation t is said to be less partial than t' if every source element of t' is also a source element of t. The source elements of a translation are exactly the domain of the translation, so possibility can be phrased as the following predicate on domains:

$$\mathrm{dom}\, t' \subseteq \mathrm{dom}\, t.$$

Coverage indicates how much of the target language is expressible as a translation of source elements. A translation t is said to be more expressive than t' if every target element of t' is also a target element of t. The target elements of a translation are exactly the range of the translation. Coverage is therefore the same as the level of surjectivity of the translation. This property can be phrased as the following predicate on ranges:

$$\mathrm{ran}\, t' \subseteq \mathrm{ran}\, t.$$

For a given translation, t, a source element can be translated into many different target elements. In that sense, the translation is ambiguous. If the translation is unique for all source elements, then it is unambiguous. Two translations over the same source and target languages can be compared to determine if one is less ambiguous than the other. We say that t is less ambignous than t' if every source element of t' has fewer possible translations in t. We represent this as the following predicate on image sets of source elements:

$$\forall es : (\mathrm{dom}\, t' \cap \mathrm{dom}\, t)$$
$$\bullet\ t(\!|\{ es \}|\!) \subseteq t'(\!|\{ es \}|\!).$$

We can combine the properties of possibility, coverage and ambignity to create a partial ordering on translations which will give a comparative measure of definedness for translations on fixed source and target languages. A translation t is said to

be more defined than t', written t $moredefined_{[S,T]}$ t', iff t provides more possible translations of the source language, covers more of the target language and is less ambiguous than t'.

$$
\begin{array}{|l}
\hline
[S, T] \\
moredefined : Translation[S, T] \leftrightarrow Translation[S, T] \\
\hline
t \ moredefined \ t' \Leftrightarrow (\quad \mathrm{dom}\, t' \subseteq \mathrm{dom}\, t \\
\qquad\qquad\qquad\qquad \wedge\ \mathrm{ran}\, t' \subseteq \mathrm{ran}\, t \\
\qquad\qquad\qquad\qquad \wedge\ \forall\, es : \mathrm{dom}\, t' \\
\qquad\qquad\qquad\qquad\quad \bullet\ t(\!|\{\ es\ \}|\!) \subseteq t'(\!|\{\ es\ \}|\!)) \\
\hline
\end{array}
$$

This relation is reflexive, antisymmetric and transitive, so $moredefined_{[S,T]}$ does indeed induce a partial ordering on the translations between source language S and target language T. This partial ordering relation is similar to a relational generalization of Manna's less-defined property of functions [104] which has been discussed by Mili and Mili and Boudriga *et al.* in their work on developing relational heuristics to program refinement [108, 26].

Definedness directly addresses the informal property of a translation's coverage, as discussed in Section 4.1. We can use this ordering as a comparative measure of the agents whose external behaviour manifests some translation in the framework. Everything else being the same, a more defined translation is preferred. However, the major use of such a definedness ordering is as a refinement ordering, which we do not fully exploit in this thesis.

4.3 Correspondence between agents

Definedness only indirectly addresses questions of ease of a translation. A more direct approach is motivated by Hutchins, Hollan and Norman [85, p. 317–318]:

> In a system built on the model-world metaphor, the interface is itself a world where the user can act, and which changes state in response to user actions. The world of interest is explicitly represented and there is no intermediary between user and world. Appropriate use of the model-world metaphor can create the sensation in the user of acting upon the objects of the task domain themselves. We call this aspect of directness *direct engagement*.

The ease of a translation between components in the framework, therefore, relies on a correspondence between their external stimuli and responses and the internal states to which they are linked in their agent representations. We will now investigate how to formalize such a correspondence between agents.

Correspondence concerns the relationship between the state and stimuli of two agents. We assume there is a relationship between the underlying state spaces of the two agents and we investigate whether the operations that can be performed on one agent are sufficiently mirrored as operations on the other agent. With respect to interactive systems, a similar feature was introduced as *state display conformance* by Harrison and Dix [73] and has been further discussed by Abowd, Dix and Harrison [2] with respect to graphical interfaces. The notion of state display conformance is very similar to the notion of data refinement (or data reification), which has been treated by He *et al.* [78] and Coenen *et al.* [37] in a general relational format. We will use the ideas of data refinement in forming our definition of correspondence.

The purpose of data refinement is to characterize how the function of an abstract description of a state machine can be adequately captured in a more concrete representation. A concrete representation is supposed to be indistinguishable from its abstract counterpart, in the sense that an outside observer interacting with the concrete representation cannot distinguish it from the abstract one. The concrete representation is then a suitable *simulation* of the abstract [76, 170]. We are not concerned here with the abstract-concrete refinement. Rather, we are concerned with the relationship that exists between the *System* and the interface (*Input* along with *Output*) of an interactive system. The observer in this case is the *User*, and we want to stipulate that operations performed on the interface which the *User* observes accurately reflect operations being performed on the *System*. In this section, we will develop a formal definition of correspondence and in Chapter 7 we will investigate further its relevance to interactive systems.

We will describe correspondence as a constraint between two agents, say A and B. Figure 4.2 gives a picture of the correspondence we intend to describe We want

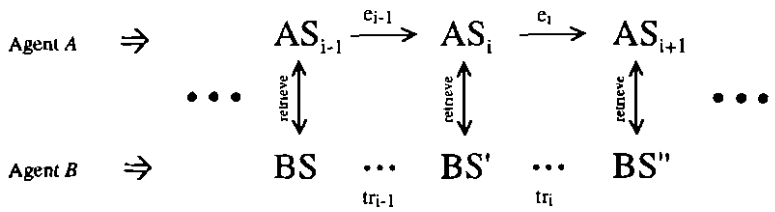

Figure 4.2: Agent correspondence

agent B to adequately mirror some functionality of agent A, i.e., we want operations performed on A to be matched by operations performed on B. The matching is done in terms of a given relationship which holds between the state descriptions of

A and B, and we will call this relation *retrieve* (of type $B.states \leftrightarrow A.states$). This relation determines from the state of B what the state of A should be, and it is understood as the interpretation A can make of the state of B.

To define this correspondence, we begin hy asserting that the initial states of A and B are related by the *retrieve* relation. Then for any state that A can attain there must have been a way for B to attain a related state. And for any future state that A can attain, B must be ahle to attain a related state. Note that we do not demand that at all times A and B are in related states. The transitions of A are what determine the states that B must eventually attain, and not the other way around.

The schema *AgentCorr* gives the formal definition motivated by the above description of correspondence.

AgentCorr

$Agent^I$

$Agent^J$

$retrieve : states^J \leftrightarrow states^I$

$inits^J \times inits^I \subseteq retrieve$

$\forall (tr_I, s_I), (tr_I \frown tr'_I, s'_I) : I^{int}_{\theta Agent^I}$
| $(s_I, s'_I) \in tr'_I \fatsemi transform^I$
• $\exists (tr_J, s_J), (tr_J \frown tr'_J, s'_J) : I^{int}_{\theta Agent^J}$
  | $(s_J, s'_J) \in tr'_J \fatsemi transform^J$
  • $\{(s_I, s_J), (s'_I, s'_J)\} \subseteq retrieve$

We define *correspond_* as a relation between agents indexed hy a retrieving relation between their state sets. The pair of agents (D, R) is in $correspond_{ret}$ if agent R corresponds to agent D as defined by the schema *AgentCorr* with retrieving relation *ret*.

$correspond\_ : (State \leftrightarrow State) \rightarrow (Agent \leftrightarrow Agent)$

$correspond_{ret} = \{\ AgentCorr$
$\quad | \quad retrieve = ret$
$\quad • \quad \theta Agent^I \mapsto \theta Agent^J\ \}$

We have given in this section a very coarse treatment of correspondence. In Chapter 8 we will see how correspondence can be treated more constructively in the refined agent model using templates. The purpose of the coustructive treatment is to formally link the empirical psychological evidence which identifies usefnl template information with the heuristic motivation behind multiagent architectures.

4.4 Predictability

Users interact with any system in order to satisfy certain goals. Fundamental to goal satisfaction is the ability to plan interaction which will achieve the goal. In a graphic drawing package, the user may want to select one or more of a collection of overlapping figures in order to perform some similar operations to them. In this case it is necessary that the user be able to predict how to select the figures of interest. *Predictability* is a property of interaction concerning the degree of confidence with which a user can determine the effect subsequent task execution will have on the achievement of the goal. Therefore, predictability is necessary for effective interaction and is a crucial measuring stick by which to gauge the usability of an interactive system. In the following, we will formalize how predictability can be expressed on agents and characterize levels of predictability within the interaction framework.

A formal treatment of predictability was first given with the PIE model by Dix and Runciman [54]. Predictability in the PIE model is defined by an unambiguous interpretation function between programs and effects. When two programs lead to the same effect, then future experimentation will betray no difference between them. Letting P represent the set of programs, E the effects set and I the interpretation function between programs and effects ($I : P \rightarrow E$), a predictable PIE satisfies the following predicate.

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

This says that the current effect is enough to predict future behaviour.

In terms of the agent model, predictability can be seen as an observable relationship between the stimuli and responses. Based on the Sufrin and He approach to classifying interactive properties as relationships between input histories and state or response histories, we can model this predictability (and other interactive properties) within the simple agent model.

Viewing an agent as another representation for a PIE, we can see that the sequences of stimuli are the programs and the history of responses are the effects. From the definition of an agent we derived an external interpretation relation, I_A^{ext}, which provides a relationship between a program (sequence of stimuli) and the resulting responses. I_A^{ext} is a relation, not a function, and so our formulation of predictability is slightly more cumbersome than for the PIE model. One program can produce more than one possible sequence of responses. Two different programs, p and q, which yield the same possible responses are said to be *externally equivalent*, and we write this as $p \overset{ext}{\approx}_A q$.

$$\_ \overset{ext}{\approx}_{\_\_} : Agent \rightarrow (\text{seq } Event \leftrightarrow \text{seq } Event)$$

$$\_ \overset{ext}{\approx}_A \_ \subseteq (\text{dom } I_A^{ext}) \times (\text{dom } I_A^{ext})$$

$$p \overset{ext}{\approx}_A q \Leftrightarrow I_A^{ext}(\!|\{p\}|\!) = I_A^{ext}(\!|\{q\}|\!)$$

Programs that are initially externally equivalent may cease to be so after the next stimulus event is received and the agent transforms itself and responds. If any such experimentation on equivalent programs betrays no such difference, then the programs are said to be *externally indistinguishable*, and we write $p \overset{ext}{\equiv}_A q$.

$$\_ \overset{ext}{\equiv}_{\_\_} : Agent \rightarrow (\text{seq } Event \leftrightarrow \text{seq } Event)$$

$$\_ \overset{ext}{\equiv}_A \_ \subseteq (\text{dom } I_A^{ext}) \times (\text{dom } I_A^{ext})$$

$$p \overset{ext}{\equiv}_A q \Leftrightarrow \forall r : \text{seq } Event \bullet (p \frown r) \overset{ext}{\approx}_A (q \frown r)$$

A predictable agent would be one in which all externally equivalent programs are externally indistinguishable. Since it is trivially true that all externally indistiu-guishable programs are externally equivalent (let $r = \langle\,\rangle$ in the definition above), predictable agents can be characterized by the equality of the relations $\overset{ext}{\approx}_A$ aud $\overset{ext}{\equiv}_A$.

$$Predictable : \mathbb{P}\ Agent$$

$$A \in Predictable \Leftrightarrow (\_ \overset{ext}{\approx}_A \_) = (\_ \overset{ext}{\equiv}_A \_)$$

In the same way that the PIE model can address varying levels of abstraction by suitable definition of the program and effects sets, we also can apply predictability at different levels within the framework. Three levels naturally arise from the framework, and we can investigate what predictability means at each of these levels. Figure 4.3 depicts the various levels we will discnss.

At the lowest level, we consider stimuli aud responses of the *System* alone. Predictability at this level we call *algorithmic predictability*. This is the simplest form of predictability possible and it is a necessary condition for the remaining levels. We assume that algorithmic predictability is satisfied so that all programs issued to the *System* which are externally equivalent are also externally indistinguishable.

At the next level, we consider the physical interface. Predictability at this level is *perceivable predictability* because it is here that contact with the *User* is possible. Perceivable predictability says that the information provided by the history of *Output* responses is enough to determine how fnture programs received by the *Input* will affect the *Output*. It is therefore possible that the *User* could predict the outcome of future input.

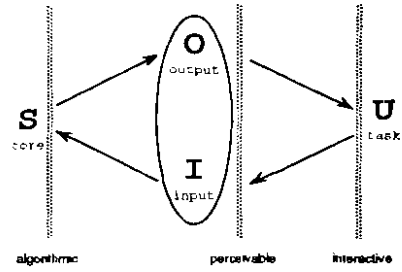

Figure 4.3: Levels of predictability

At the highest level, we move beyond the world of keystrokes and displays to include how the *User* observes the *Output* and understands the *Input*. *Interactive predictability* means that that which the *User* understands of the history of *Output* responses is enough to determine the effect of future tasks. Many systems can satisfy perceivable predictability and yet remain unusable because of the way the *Output* responses are observed by the *User*.

One feature of the agent model which we rely on is its compositionality. This will be further developed in Chapter 5. For now, we will assume that components of the framework, which are defined as agents, and the translations, which are external descriptions of agents, can be composed to form more complex agents. If we combine all of the components and translations of the interaction framework except for the *User* we have a user's model of the system, or, more precisely, the designer's model of the user's model of the system, depicted in Fignre 4.4. The external interpretation

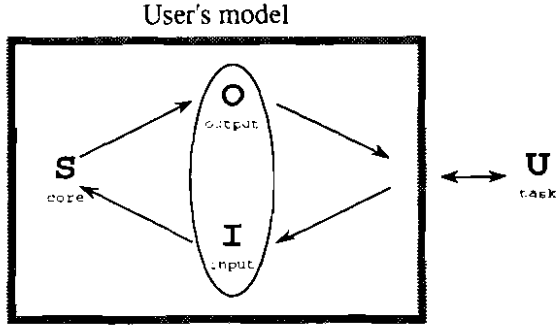

Figure 4.4: The user's model of the system

relation for this agent, which we refer to as $I^{ext}_{User's\ model}$, relates information at the level of how *Input* stimnli and *Output* response are nnderstood by the *User*. Hutchins, Hollan and Norman refer to this level of information as the meaning of the input and output interface languages[85].

A colloquial expression for this interactive predictability would be

> *What you have done up to now determines what you will see from now on.*

There are two separate aspects of this statement to consider, what is seen aud what is remembered. $I^{ext}_{User's\ model}$ makes explicit the assumptions about what is seen and how the *User* understands the responses from the *Output*. For example, when using a calculator, it may be a safe assumption that nnmbers displayed are interpreted in base 10, so imagine the surprise and consternation when the user enters 9+4 and gets back the response D! What is remembered is important because sometimes the order of stimuli and responses contaius information crucial for predictability. For example, in an interactive drawing package, the order in which overlapping objects are created can determine the effect of clicking the mouse button to select objects when the selection cursor is located in the overlapping region.

Predictability as defined above assumes both perfect memory aud equivalence over the entire history of the interaction. This is not satisfactory since users forget and similarity of future responses does not depend on exact similarity of past responses. Predictability is a feature of an interactive system which enhances the interaction. Formalizing what it means to be predictable highlights the cost associated in order to realize the benefits of predictability. Our earlier definition has too high a cost, and so we would like to investigate ways of reducing that cost. One way to reduce the cost is by revising the definition of equality so that it is not over entire response histories bnt over some subset and only on particular attributes associated with the responses as well.

4.5 Nondeterminism

Predictability only cousidered the external behavionr of an agent. The responses from an agent occur after the state transition fired by the stimnlus. There is more information available at the state level than at the response level. We can apply the method used for predictability on responses to formalize an internal version of predictability. The internal interpretation relation, I^{int}_A relates programs to the possible states the agent can be in after participating in the program. Internally equivalent programs result in the same possible states. Internally indistinguishable programs will betray no state difference.

$$
\begin{array}{|l}
\_ \overset{int}{\approx} \_ ,\\
\_ \overset{int}{\equiv} \_ \_ : \mathit{Agent} \rightarrow (\operatorname{seq} \mathit{Event} \leftrightarrow \operatorname{seq} \mathit{Event}) \\
\hline
\_ \overset{int}{\approx}_A \_ \subseteq (\operatorname{dom} I_A^{int}) \times (\operatorname{dom} I_A^{int}) \\
\_ \overset{int}{\equiv}_A \_ \subseteq (\operatorname{dom} I_A^{int}) \times (\operatorname{dom} I_A^{int}) \\
p \overset{int}{\approx}_A q \Leftrightarrow I_A^{int}(\!|\{p\}|\!) = I_A^{int}(\!|\{q\}|\!) \\
p \overset{int}{\equiv}_A q \Leftrightarrow \forall r : \operatorname{seq} \mathit{Event} \bullet (p \frown r) \overset{int}{\approx}_A (q \frown r)
\end{array}
$$

A *deterministic* system is internally predictable, so that internally equivalent programs are also internally indistinguishable.

$$
\begin{array}{|l}
\mathit{Deterministic} : \mathbb{P}\, \mathit{Agent} \\
\hline
A \in \mathit{Deterministic} \Leftrightarrow (\_ \overset{int}{\approx}_A \_) = (\_ \overset{int}{\equiv}_A \_)
\end{array}
$$

The state holds enough information to determine future internal behaviour.

We have modelled the behaviour of an agent as a relation which reflects two types of nondeterminism—true randomness or 'don't care' nondeterminism. True randomness is very rare in computers. 'Don't care' nondeterminism comes from abstracting away details that will be present in any implementation but do not matter for the purpose of the current description. Don't care nondeterminism is very common and encouraged in abstract specification. It is the main reason we have modelled behaviour as a relation rather than as a function. The definition of deterministic above essentially rules out internal randomness. It does not rule out nondeterminism that arises from abstraction.

Another form of nondeterminism arises from how the user perceives the system. To a naive user even the simplest of interactive systems is nondeterministic. An example used by Dix [48, 49] and Thimblehy [161] is of a prime number generator. Even though the algorithm for deriving the sequence of increasing primes is deterministic, if the user does not know that the system is producing primes, the sequence `2,3,5,7,11,...` may appear somewhat random. It is only through experience interacting with the system that the user can begin to understand its deterministic functionality. And this user-perceived nondeterminism arises naturally even for an expert, as uncertainty about the past interaction history leads to uncertainty of the current state. Therefore, even though an agent's behaviour may satisfy the definition of deterministic above, it can still seem nondeterministic to its observer.

Equivalence was based on all state information. Only part of the state information may be important for a given task, and so we may want to analyze whether the agent was deterministic relative to that part of the state. To do this, we will need a way to highlight specific information about the state. The simple agent model does

not allow us to constructively define such a restricted view, and so we define the state of the refined agent model in Chapter 5 in terms of attribute-value mappings. Then in Chapter 7, we define restrictions on agents by means of templates, which are simply formed as subsets of the agent's state attributes.

4.6 Synthesis

Predictability assumes that the user has some knowledge of how the system works, i.e., that the user has internalized a model of how the system works. Expert users will have a more complete internalized model than novices, which reflects their greater experience. Most users, expert or novice, will continually be presented with situations which are new to them and for which their model is not able to provide enough information to predict future outcomes of subsequent input. In these cases, the user will experiment with the system by offering some input and trying to detect the effects which result, hence building up more information in their own model. We call this process of internalizing input-output relationships *synthesis*. Synthesis is complementary to predictability. A predictable system is one in which it would be possible for the user to internalize a model that would be of benefit to future interaction. Predictability is done by reasoning forward in time based on information available at present. Synthesis involves reasoning back in time in order to determine how the present information was achieved from the past. Figure 4.5 shows how predictability and synthesis relate to the interaction framework.

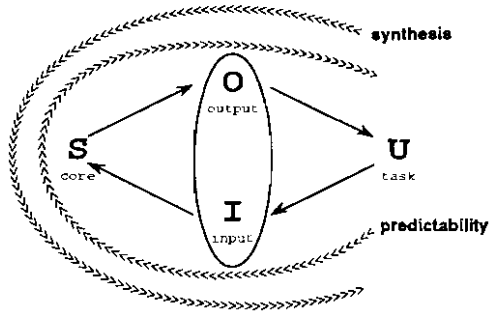

Figure 4.5: Predictability and synthesis

The observer of an agent only has access to the stimulus-response behaviour of the agent. Since the stimuli cause state transitions which in turn determine the responses, effective synthesis depends on changes to the underlying state being

reflected in the responses. If a program changes the underlying state, then the subsequent responses should indicate this. In order for the observer to have the chance of correctly synthesizing the effect of the previous input, the response should occur before any other changes are made to the internal state. An *honest* agent will satisfy the property that externally equivalent programs are also internally equivalent. A weaker property would only require that externally indistinguishable programs be internally equivalent; such agents are deemed *trustworthy*[4].

$$
\begin{array}{|l}
Honest, Trustworthy : \mathbb{P}\, Agent \\
\hline
A \in Honest \Leftrightarrow \_ \overset{ext}{\approx}_A \_ \subseteq \_ \overset{int}{\approx}_A \_ \\
A \in Trustworthy \Leftrightarrow \_ \overset{ext}{\equiv}_A \_ \subseteq \_ \overset{int}{\approx}_A \_
\end{array}
$$

A good example of the difference between honesty and trustworthiness can be seen with search and replace facilities in most text editors. A single search for a string of text followed by the replacement with another string of text is honest, for the replaced text is usually displayed on screen right where the old text was located. Most global search and replace facilities do not display all of the changes made, leaving the user to confirm the changes made by subsequent browsing of the text. Global facilities such as this can only ever be as good as trustworthy.

4.7 Consistency

Consistency is prominent in the HCI and ergonomic literature [113, 142]. It is put forth as a cause for increased learnability of a system [142, 93, 32]. In reading different accounts of consistency, there is clearly a debate about what exactly is meant by consistent at the psychological level (witness, for example the debate on consistency between Grudin [64] and Reisner [136]). Such informal arguments highlight even more the importance of a formal framework in which different formulations of consistency can be compared and contrasted.

As an example of how we can formalize consistency, we will take a simple definition. Our informal definition of consistency is that the same input in similar situations has similar effects. This is a generalization of predictability in which the same input in the same situation has the same effect. The criteria for deciding similarity are crucial, just as the criteria for equivalence were for predictability. And as was done for predictability, we can discuss consistency internally and externally. Consistency of an agent is defined with respect to the similarity criterion,

[4]Sufrin and He originally used the names honest and trustworthy to apply to individual programs which satisfy conditions somewhat similar to those described here. Their definition, however, demanded that programs which were not indistinguishable internally were either not equivalent (for honesty) or not indistinguishable (for trustworthy) externally.

which is modelled as an equivalence relation. Internal consistency depends on a similarity relation, say $\sim$, which is an equivalence relation on the states of tbe agent. The state transition relation associated to any program must then respect the equivalence classes of $\sim$.

$$
\begin{array}{|l}
IntConsistent : \mathbb{P}(Agent \times (State \leftrightarrow State)) \\
\hline
(A, \sim) \in IntConsistent \Leftrightarrow \\
\quad (\ \text{equivalence}(\sim) \\
\quad \wedge \sim \subset A.states \times A.states \\
\quad \wedge \forall p : \text{seq } Event;\ E1, E2 : classes(\sim) \\
\quad\quad \bullet\ E1 \lhd (transformExtend\ p) \subseteq E2
\end{array}
$$

A similar definition applies for external consistency, with the only difference being that the similarity relation is defined on the histories of responses for the agent. The interesting question to ask now is whether for an externally consistent agent A with respect to response similarity criterion $\sim$, is there a corresponding state similarity criteriou $\sim'$ such that A is internally consistent with respect to $\sim'$? Within the interaction framework, it would be possible for the *User* to detect the external consisteucy of the interface and it would be useful if that consistency mapped down to some consistency internal to the *System* which related to the domain of applicatiou. Likewise, the *User* may rely on some internal consistency of the *System* which would then be expected to be manifested externally at the interface.

Whereas this definition of consistency is simple to express, we must exercise caution in understauding what it actually is saying. First of all, we had to assume that there was a way to partition the state or responses by some nontrivial similarity criterion. And we assume that this similarity criterion is relevant to the observer for external consistency. If such similarity criteria exist, then an advantage of this definition of consistency is that it is cumulative. By this we mean that if we have two similarity criteria then their conjunction is also a similarity criterion (though there again is no guarantee that it will be relevant to the observer). Therefore, consistency as we have defined it also has many levels of applicatiou. A word of caution. however, since compounded equivalence relations will tend to have many more and smaller equivalence classes, until the compounded similarity criterion gives no more information than the trivial one.

Another possible disadvantage of this definition is its application to all possible programs. It seems more realistic to assume that consistency properties will be relevant for subsets of programs. Finally, consistency is defined with respect to the similarity criteria. Therefore, an agent consistent by one similarity criterion may be iuconsistent by another, suggesting a trade-off to be made during design.

We have only cousidered one formalization of consistency here. An obvious other consistency notion would have similar input in the same situation resulting

in similar behaviour, but we have not investigated this possibility yet.

4.8 Conclusions

In this chapter we have provided both qualitative and quantitative evidence for properties of interactive system which enhance the interaction between user and system. The qualitative accounts have appealed mainly to a common sense understanding of how the various translations in the framework impact on the overall effectiveness of interaction. Two important features of any translation were identified as its coverage or possibility of converting one language to another and the ease with which this conversion can occur. The interaction framework provided a mechanism for applying heuristics for effective interaction uniformly.

Though we strongly urge the use of heuristic reasoning in the design and analysis of interactive systems, our main emphasis in this thesis is on transferring that heuristic knowledge onto a more rigorous software engineering platform based on formal methods. The attempts to formalize various interactive properties in terms of the simple agent model have shown that the formalism can capture the intended meaning of those properties. But more importantly, by making our definitions of the properties such as predictability and consistency formal, we can more easily see what it is about those properties that we have *not* captured in the formalism.

For example, we can see that our definition of predictability assumes we can judge equality over complete response histories for an agent, and this seems too stringent a requirement on the observer of the agent, especially when that observer is a user. We require a way to express predictability so that it embodies more realistic coustraints on the observer.

Though the agent model breaks open the black box of the PIE model somewhat, allowing expression of properties such as correspondence and honesty, it is clear that we have not opened the box wide enough. We still have not provided an easier way to describe the behaviour of an agent in order to accommodate proof of satisfaction of various interactive properties. We have only slightly motivated the need for a compositional agent model, and this needs further investigation.

One of the advantages of the red-PIE model and of Sufrin and He's model of interactive processes was that they both distinguished that which is discernible by the user (the display) from that which the user wants to achieve from the interaction (the result). We have made light of this distinction in the simple agent model. Displays are immediate and ephemeral indications from a system and we have equated those with responses of an agent. Results are more permanent and we have equated them with the underlying state of the agent. This is not satisfactory because displays are more like state machines than a sequence of responses would indicate and the state of an agent may contain information that is not relevant to

State-Response relationship	Interactive Property
$(\_ \overset{ext}{\approx}_A \_) = (\_ \overset{ext}{\equiv}_A \_)$	predictability
$(\_ \overset{int}{\approx}_A \_) = (\_ \overset{int}{\equiv}_A \_)$	deterministic
$(\_ \overset{ext}{\approx}_A \_) \subseteq (\_ \overset{int}{\approx}_A \_)$	honesty
$(\_ \overset{ext}{\equiv}_A \_) \subseteq (\_ \overset{int}{\approx}_A \_)$	trustworthiness

Table 4.1: Interactive properties expressed as state-response relationships

the end result from the user's point of view. We will need a way to more faithfully represent these crucial notions of display and result.

We have adopted Sufrin and He's approach to classification of interactive properties in terms of equivalence and indistinguishability because the structure of the classification suggests the possible properties that can be expressed. Table 4.1 lists the properties we have discussed in terms of equivalence and indistinguishability of state and response (internal and external). A similar classification approach which takes into account result and display information will be the subject of Chapter 7.

Thus we have set the agenda for the remainder of this thesis. We begin with a refinement of the agent model in Chapter 5 followed by a definition of a design language for describing agents in Chapter 6. The state of the agent will be described in terms of attributes and so we will be able to extract partial information, or templates on the state. We will then show how the refined agent model allows the expression of further interactive properties in Chapter 7. The introduction of attributes and templates will lead to a more direct treatment of the result-display relationships that will take a similar form to the state-response relationships. The description of agents in the refined model is more closely linked to empirical psychological evidence of how users understand the tasks the agents are designed to support. The interactive properties will also be based on that task-specific evidence, providing for a user-centred methodology for the design and analysis of an interactive system. In Chapters 7 and 8 we will show how the methodology is implied by the description of further interactive properties and the investigation of heuristic multiagent architectures.

Chapter 5

Refining the agent model

In preceding chapters we have discussed an overall framework for the description of an interactive systems and how it provides a basis for the description of general properties of interaction expressed within a formal model. The purpose of this chapter is to refine the model of the agent used in the previous chapters. The next chapter then provides a design language for agents.

The agent model we present in this thesis is directly motivated by the formal presentation of the interaction framework. We desire a language that will allow the discussion of interactive properties and with which it will be possible to describe a complete system. The formal interaction framework provided a means for a precise discussion of interactive properties, but it did not allow a realistic description of a complex system. What is required is a compositional model, and this is what the refined agent model is intended to provide.

The refinement of the agent model in this chapter is a necessary digression. We have seen the need for more detail of the structure of an agent in order to describe interactive properties more closely related to the way a user perceives interaction with respect to the goals of tasks. The refinement is also necessary from a software engineering point of view. To bring HCI considerations into earlier stages of design it is not sufficient to merely provide an abstract mathematical model for describing interactive properties. The model must be expressible by some design language and accompanied by some method for directing the description. In this chapter, we address the design issue by providing a compositional agent model.

Overview of chapter

In Section 5.1 we will formulate the requirements placed on our agent model for its use in the design and analysis of interactive systems. One of the major requirements is that the refined model supports a modular description of complex agents by the composition of smaller and simpler agents.

The description of the refined agent model is separated into three stages representing the internal state-based specification, the external event-based specification and the communication specification, which links events to internal operations. Hence, the outline for the refined agent model is as below.

Agent
InternalSpec
ExternalSpec
Communication

In Sections 5.2, 5.3 and 5.4 we will address separately the internal, external and communication specifications of the refined agent model. In each section, the model for the particular specification will be given and then it will be shown how separate specifications can be composed associatively. There are two ways we will compose agents, by interleaving and by synchronization. In Section 5.5, we will unite the separate specifications and show how interleaving and synchronization are defined between agents.

In Section 5.6, we will show how the refined agent model relates to the simpler model of Chapter 2 by defining the internal and external interpretation functions.

5.1 Requirements for agents

An agent has state and a means for communicating stimulus and response events. A pictorial view of an agent is given in Figure 5.1. The stimulus and response events

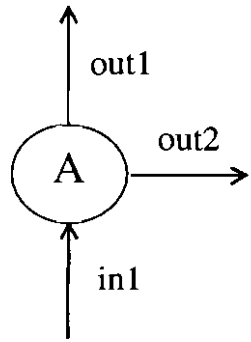

Figure 5.1: Pictorial representation of an agent

are communicated along input and output channels. There are several requirements

on the refined model which motivate its development in this chapter and the next. Here we will discuss requirements relating to compatibility with the simple agent model and compositionality.

We have already given a simple definition of an agent in Chapter 2 which is adequate for expressing a variety of interactive properties at the abstract level. The new model must be consistent with the older model. The two most important features of the simpler model were the derived internal and external interpretation relations, I_A^{int} and I_A^{ext}, so we must be able to derive similar relations with the new model. It turns out that though we can express the external interpretation function, I_A^{ext}, we will not use it in formulating properties relating result and display in Chapter 7.

In Chapter 4, we assumed that agents could be composed to form more complex agents. We will need to demonstrate to what degree the refined agent model allows the construction of complex agents through the combination of existing agents. Given two agents, A and B, there will be two ways of combining them to achieve a third agent.

We may want to treat A and B as independent agents that can only affect each other's private state by means of synchronous message passing along common communication channels. The agent derived by this synchronous composition we will denote by $compose_{sync}(A, B)$. There are various conditions that are imposed on A and B so that they may be synchronously composed. They must have separate state spaces with no common attributes. Any common channels must be an input for one agent and an output for the other. The common channels become synchronized, meaning that communications that occur along them must be participated in by both component agents simultaneously. The synchronized channel is no longer visible outside the composed agent. Figure 5.2 is a graphic depiction of synchronous composition.

We may want to treat A and B as dependent agents. In this case, some part of their internal states is shared between them. Such dependent composition we describe by interleaving their descriptions to obtain another agent, denoted by $compose_{int}(A, B)$. There are conditions we impose on interleaving composition as well. The separate agent's states may be defined over common attributes and they may have common input or output channels. The effect of operations defined for both agents is constrained such that they only alter the values of common attributes. Figure 5.3 graphically depicts this composition by interleaving. The overlapping circles are meant to indicate shared state information. Note that interleaved agents cannot communicate via synchronous channels, so an input for one agent cannot be an output for the other agent. In this sense, the agents are unlinked.

In Section 5.5, we present two main theorems concerning the associativity of synchronons and interleaving composition. The proof of these theorems is dealt with in terms of smaller theorems on the associativity of composition for the in-

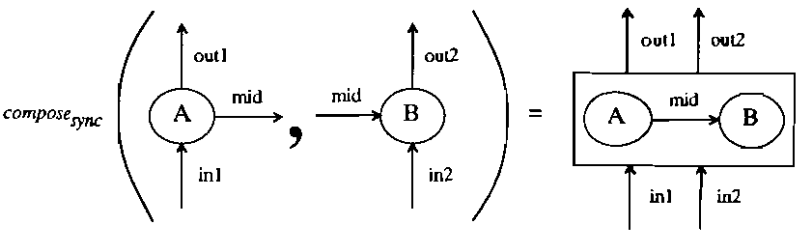

Figure 5.2: Synchronous composition of independent agents

ternal, external aud communication specifications of the previous sections. The associativity of composition will allow any number of agents to be composed either all synchronously or all by interleaving without worrying about the order of the composition.

5.2 Internal specification

An agent has its own private and persistent state. The possible set of states for an agent is described by the values associated to some finite set of attributes. We assume a set of attribute names and a universal set of values. A state is a finite function from attributes to values.

$$[\mathcal{A}, \mathcal{V}]$$
$$State == \mathcal{A} \nrightarrow \mathcal{V}$$

A *state set* is a set of states, all of whose elements are defined over the same set of attributes. In addition, the attributes of a state set have *types* associated with them. A type is a nonempty subset of $\mathcal{V}$. The values of an attribute are restricted to beiug in that attribute's type.

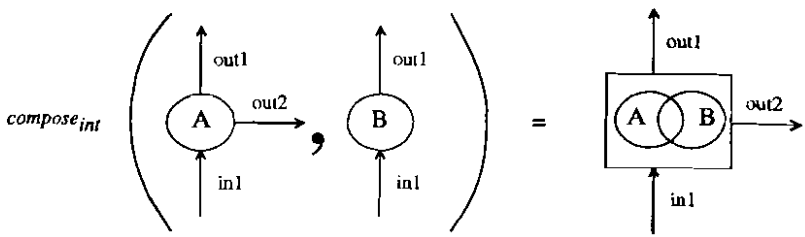

Figure 5.3: Interleaving composition of dependent agents

Stateset

$attribs : \mathbb{F}\, \mathcal{A}$

$type : \mathcal{A} \pfun \mathbb{P}_1\, \mathcal{V}$

$states : \mathbb{P}_1\, State$

$attribs = \mathrm{dom}\ type$

$\forall s : states; a : attribs \bullet s(a) \in type(a)$

$states \subseteq (attribs \fun \mathcal{V})$

The internal specification of an agent gives the state set along with operations defined on that state set. Each operation is labelled by a *message*, obtained from a set of all possible messages. The internal specification is completed by giving the set of initial states.

$[Message]$

InternalSpec

$Stateset$

$inits : \mathbb{P}_1\, State$

$operations : Message \pfun (states \rightarrow states)$

$messages : \mathbb{P}\, Message$

$inits \subseteq states$

$messages = \mathrm{dom}\ operations$

Combining internal specifications

Two separate internal specifications can be combined to create a new internal specification. One constraint on the combination concerns the separate state sets—they must agree over all common attributes. If s_1 is a state of one internal specification and s_2 is a state of the other internal specification, and the values of s_1 and s_2 agree over their common attributes, i.e., s_1 and s_2 are *compatible*, then they define a state in the combined internal specification. A simple way to express this compatibility constraint takes advantage of the functional representation of a state and says that two states are compatible if overriding one by the other is equal to the functional union of the two.

$$compatible : State \leftrightarrow State$$

$$(s_1, s_2) \in compatible \Leftrightarrow s_1 \oplus s_2 = s_1 \cup s_2$$

The schema *JoinStatesets* characterizes the combination of two state sets to form a third state set whose elements are the union of such compatible pairs of states. We have stipulated that the state sets are nonempty, so state sets which are to be joined must have at least one pair of compatible states. We choose the name join because it is suggestive of the similar operation for relational databases. Joining is also similar to schema signature combination in Z, where components with the same name from two different schemas must be of the same type and take on the same values [153]. Note that the I and J are used in the following schema to decorate the argument state sets and $'$ is used to decorate the state set resulting from the join.

JoinStatesets

$$Stateset^I$$
$$Stateset^J$$
$$Stateset'$$

$$\forall a : attribs^I \cap attribs^J \bullet type^I(a) = type^J(a)$$
$$\exists s_I : states^I;\ s_J : states^J \bullet (s_I, s_J) \in compatible$$
$$attribs' = attribs^I \cup attribs^J$$
$$type' = type^I \cup type^J$$
$$states' = \{\ s_I : states^I;\ s_J : states^J \mid (s_I, s_J) \in compatible \bullet s_I \cup s_J\ \}$$

For convenience, we can represent the above schema definition as a binary operation *join* on *Stateset*. We will frequently make use of this mechanical technique

for characterizing a combination via a schema and then converting it to a binary operation.

$$
\begin{array}{|l}
join : (Stateset \times Stateset) \nrightarrow Stateset \\
\hline
join = \{\ JoinStatesets \\
\qquad \bullet\ (\theta Stateset^{I}, \theta Stateset^{J}) \mapsto \theta Stateset^{\prime}\ \}
\end{array}
$$

As we described in Section 5.1, we will want to compose several agent descriptions and we want that composition to be associative. The proof of the associativity of the overall agent composition functions is distributed throughout this and the next two sections. The first theorem below shows that joining of states sets satisfies associativity.

Theorem 5.1

$$
\begin{array}{l}
\forall S1, S2, S3 : Stateset \\
\quad |\ \ (S1, S2) \in \mathrm{dom}\, join \\
\qquad \wedge\ (join(S1, S2), S3) \in \mathrm{dom}\, join) \\
\quad \bullet\ join(join(S1, S2), S3) = join(S1, join(S2, S3))
\end{array}
$$

PROOF OF THEOREM 5.1:

The proof proceeds in two stages. First, under the conditions of the hypothesis, namely that

$$
\begin{array}{l}
(S1, S2) \in \mathrm{dom}\, join \\
\wedge\ (join(S1, S2), S3) \in \mathrm{dom}\, join
\end{array}
$$

we must show that

$$
\begin{array}{l}
(S2, S3) \in \mathrm{dom}\, join \\
\wedge\ (S1, join(S2, S3)) \in \mathrm{dom}\, join
\end{array}
$$

After proving that $(S1, join(S2, S3)) \in \mathrm{dom}\, join$, we must show the equality of the different joins. Both stages of this proof rely on a lemma on state compatibility which says that if the union of two compatible states is compatible with a third state, then each of those states is compatible separately with the third state. Full details of the proof of this lemma and the stages of this theorem are provided in Appendix B.

◇ END OF PROOF OF THEOREM 5.1

To complete the definition of composition of internal specifications, we must show how the operations of two separate internal specifications are coalesced to derive the operations of the combined internal specification. The explanation behind this derivation is relatively straightforward. The messages of the new internal

specification fall into one of three sets—messages nnique to one specification, or unique to the other, or common to both. If a message is nnique to one of the internal specifications, then it is lifted to an operation which is consistent with the original specification and behaves like the identity transition elsewhere. If a message is common to both internal specifications, then it is lifted to an operation which is consistent with both of the original specifications. The derivation of the operations set reflects the three mntually exclusive conditions on messages of the combined internal specification, and hence is a bit large. We could have been more concise with the description of *operations*, but this large definition mirrors the way we regard the operations set in the associativity proofs which follow.

The schema *CombineInt* defines how two internal specifications can be combined to form a third internal specification.

CombineInt

$$InternalSpec^I$$
$$InternalSpec^J$$
$$InternalSpec'$$
$$JoinStatesets$$

$$\exists\, s_0^I : inits^I;\ s_0^J : inits^J \bullet (s_0^I, s_0^J) \in compatible$$

$$inits' = \{\, {}^I_0 : inits^I;\ s_0^J : inits^J \mid (s_0^I, s_0^J) \in compatible \bullet s_0^I \cup s_0^J \,\}$$

$$messages' = messages^I \cup messages^J$$

$$\forall\, m : messages^I - messages^J$$
$$\bullet\ operations'(m) =$$
$$\{\, s, s' : states'$$
$$\mid\ (\ (attribs^I \lhd s, attribs^I \lhd s') \in operations^I(m)$$
$$\wedge\ (attribs^I \lhd\!\!\!\!- s, attribs^I \lhd\!\!\!\!- s') \in \mathrm{id}(KnotIattribs \nrightarrow \mathcal{V}))$$
$$\bullet\ s \mapsto s' \,\}$$

$$\forall\, m : messages^J - messages^I$$
$$\bullet\ operations'(m) =$$
$$\{\, s, s' : states'$$
$$\mid\ (\ (attribs^J \lhd s, attribs^J \lhd s') \in operations^J(m)$$
$$\wedge\ (attribs^J \lhd\!\!\!\!- s, attribs^J \lhd\!\!\!\!- s') \in \mathrm{id}(KnotJattribs \nrightarrow \mathcal{V}))$$
$$\bullet\ s \mapsto s' \,\}$$

$$\forall\, m : messages^I \cap messages^J$$
$$\bullet\ operations'(m) =$$
$$\{\, s, s' : states'$$
$$\mid\ (\ (attribs^I \lhd s, attribs^I \lhd s') \in operations^I(m)$$
$$\wedge\ (attribs^J \lhd s, attribs^J \lhd s') \in operations^J(m))$$
$$\bullet\ s \mapsto s' \,\}$$

where $KnotIattribs = attribs' - attribs^I$
$KnotJattribs = attribs' - attribs^J$

The first predicate stipulates that one pair of compatible states must be from the initial states of each, to ensure that there is an initial state of the composed specification. The last three predicates cover the three cases for messages of the new internal specification.

The schema *CombineInt* describes the most general combination of internal specifications we will need. This composition can be represented as a function derived from the schema above. Given two internal specifications, which satisfy the constraints of *CombineInt* above, *Icompose* yields the unique internal specification which is their combination.

$$Icompose : (InternalSpec \times InternalSpec) \nrightarrow InternalSpec$$

$$Icompose = \{\ CombincInt \bullet (\theta InternalSpec^I, \theta InternalSpec^J) \mapsto \theta InternalSpec'\ \}$$

When interleaving agents, associativity of *Icompose* is dependent upon how the separate agents are defined to behave over any common messages. We stipulate that common messages can only change the values of shared attributes. In practice, this constraint will be satisfied because the common messages between interleaved agents will only be output messages defined to be the identity transition on all attributes. This general condition of message compatibility is characterized in the following schema. We express it mathematically by saying that the operations associated to common messages do not affect attributes not in common between the agents.

MessageCompatible

$$InternalSpec^I$$
$$InternalSpec^J$$
$$notIJ : \mathbb{P}\,\mathcal{A}$$

$$notIJ = (attribs^I \cup attribs^J) - (attribs^I \cap attribs^J)$$

$$\forall\ m : messages^I \cap messages^J;\ (s_I, s_I') : operations^I(m);\ (s_J, s_J') : operations^J(m) \bullet (\ (notIJ \lhd s_I, notIJ \lhd s_I') \in \mathrm{id}(notIJ \nrightarrow \mathcal{V}) \land (notIJ \lhd s_J, notIJ \lhd s_J') \in \mathrm{id}(notIJ \nrightarrow \mathcal{V}))$$

$$messagecompatible : InternalSpec \leftrightarrow InternalSpec$$

$$messagecompatible = \{\ MessageCompatible \bullet \theta InternalSpec^I \mapsto InternalSpec^J\ \}$$

In the case of synchronous composition, associativity is satisfied because the separate state sets are forced to be disjoint. The associativity conditions for the composition of internal specifications are summarized in the following two theorems. Theorem 5.2 treats the case for interleaving composition and Theorem 5.3 covers the case for synchronous composition. The proofs are only outlined; full proofs such as those provided in Appendix B could be provided.

Theorem 5.2 *Message compatible internal specifications can be composed associatively, i.e.,*

$$
\begin{array}{l}
\forall I1, I2, I3 : InternalSpec \\
\quad | \quad (\quad (I1, I2) \in \mathrm{dom}\, Icompose \\
\qquad \wedge\ (Icompose(I1, I2), I3) \in \mathrm{dom}\, Icompose \\
\qquad \wedge\ \{(I1, I2), (I2, I3), (I1, I3)\} \subseteq messagecompatible) \\
\quad \bullet \quad Icompose(Icompose(I1, I2), I3) = Icompose(I1, Icompose(I2, I3))
\end{array}
$$

PROOF OF THEOREM 5.2:

Theorem 5.1 allows us to prove that if the first two predicates in the hypothesis are satisfied, then the composition can be formed in either order and the state sets and initial states are equivalent. We examine the two cases for messages, ones that are not shared and ones that are.

Case 1: The message is not shared by any two of $I1, I2, I3$.

There are three subcases to investigate, but the argument is the same for each. Suppose, for example, the message belongs to $I1$ and not $I2$ or $I3$. We have the operations defined as below. We adopt a shorthand notation, so, for example,

$$operations^{I(12)3}$$

will stand for

$$Icompose(Icompose(I1, I2), I3).operations$$

The proof proceeds. For any message $m : messages^{I1}$, we have

$$
\begin{array}{l}
operations^{I(12)3}(m) = \\
\{\ s, s' : states^{I(12)3} \\
\quad | \quad (\quad (attribs^{I12} \lhd s, attribs^{I12} \lhd s') \in operations^{I12}(m) \\
\qquad \wedge\ (attribs^{I12} \lhdd s, attribs^{I12} \lhdd s') \\
\qquad\qquad \in \mathrm{id}(attribs^{I3} - attribs^{I12} \nrightarrow \mathcal{V})) \\
\quad \bullet \quad s \mapsto s'\ \}
\end{array}
$$

By definition of $operations^{I12}$, and since $m \in message^{I1}$ and Theorem 5.1 tells us that the state sets are equivalent, the above set comprehension is equivalent to the following.

$$
\begin{array}{l}
\{\ s, s' : state^{I1(23)} \\
\quad | \quad (\quad (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\
\qquad \wedge\ (attribs^{I2} \lhd attribs^{I1} \lhdd s, attribs^{I2} \lhd attribs^{I1} \lhdd s') \\
\qquad\qquad \in \mathrm{id}(attribs^{I2} - attribs^{I1} \nrightarrow \mathcal{V}) \\
\qquad \wedge\ (attribs^{I12} \lhdd s, attribs^{I12} \lhdd s') \\
\qquad\qquad \in \mathrm{id}(attribs^{I3} - attribs^{I12} \nrightarrow \mathcal{V})) \\
\quad \bullet \quad s \mapsto s'\ \}
\end{array}
$$

And the last two predicates can be collapsed to obtain the following.

$$\begin{array}{l}\{\ s, s' : states^{I1(23)} \\ \mid\ (\ \ (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\ \quad \wedge\ (attribs^{I1} \lhdd s, attribs^{I1} \lhdd s') \in \mathrm{id}(attribs^{I23} - attribs^{I1} \nrightarrow \mathcal{V})) \\ \bullet\ \ s \mapsto s'\ \}\end{array}$$

This final set comprehension is $operations^{I1(23)}(m)$, as desired. A similar proof holds in the other cases, and so we have shown that the operation sets defined are equivalent.

Case 2: The message is shared by 2 or more of I1,I2,I3.

Again, there are several subcases, but each is argued similarly. Assume, for example, that m is a message for I1 and I3, but not I2. The operations are defined as

$$\begin{array}{l}operations^{I(12)3}(m) = \\ \{\ s, s' : states^{I(12)3} \\ \mid\ (\ \ (attribs^{I12} \lhd s, attribs^{I12} \lhd s') \in operations^{I12}(m) \\ \quad \wedge\ (attribs^{I3} \lhd s, attribs^{I3} \lhd s') \in operations^{I3}(m)) \\ \bullet\ \ s \mapsto s'\ \}\end{array}$$

By definition of $operations^{I12}$, and since $m \in message^{I1}$ and Theorem 5.1 tells us that the state sets are equivalent, the above set comprehension is equivalent to the following.

$$\begin{array}{l}\{\ s, s' : states^{I1(23)} \\ \mid\ (\ \ (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\ \quad \wedge\ (attribs^{I2} \lhd attribs^{I1} \lhdd s, attribs^{I2} \lhd attribs^{I1} \lhdd s') \\ \quad\ \ \in \mathrm{id}(attribs^{I2} - attribs^{I1} \nrightarrow \mathcal{V}) \\ \quad \wedge\ (attribs^{I3} \lhd s, attribs^{I3} \lhd s') \in operations^{I3}(m)) \\ \bullet\ \ s \mapsto s'\ \}\end{array}$$

By assumption, $operations^{I1}(m)$ and $operations^{I3}(m)$ have no effect on attributes not in common, so the middle clause can be rewritten, substituting $I3$ for $I1$, to obtain

$$\begin{array}{l}\{\ s, s' : states^{I1(23)} \\ \mid\ (\ \ (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\ \quad \wedge\ (attribs^{I2} \lhd attribs^{I3} \lhdd s, attribs^{I2} \lhd attribs^{I3} \lhdd s') \\ \quad\ \ \in \mathrm{id}(attribs^{I2} - attribs^{I3} \nrightarrow \mathcal{V}) \\ \quad \wedge\ (attribs^{I3} \lhd s, attribs^{I3} \lhd s') \in operations^{I3}(m)) \\ \bullet\ \ s \mapsto s'\ \}\end{array}$$

This, in turn can be collapsed, by the definition of $operations^{I23}$.

$$\begin{array}{l}\{\ s, s' : states^{I1(23)} \\ \mid\ (\ (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\ \quad \wedge\ (attribs^{I23} \lhd s, attribs^{I23} \lhd s') \in operations^{I23}(m)) \\ \bullet\ \ s \mapsto s'\ \}\end{array}$$

This last set comprehension is eqnivalent to $operations^{I1(23)}$, as desired.

◇ END OF PROOF OF THEOREM 5.2

Theorem 5.3 *Internal specifications with no common attributes can be composed associatively, i.e.,*

$$\begin{array}{l}\forall\, I1, I2, I3 : InternalSpec \\ \mid\ (\ \text{disjoint}\ \langle I1.attribs, I2.attribs, I3.attribs\rangle \\ \quad \wedge\ (I1, I2) \in \text{dom}\ Icompose \\ \quad \wedge\ (Icompose(I1, I2), I3) \in \text{dom}\ Icompose) \\ \bullet\ \ Icompose(Icompose(I1, I2), I3) = Icompose(I1, Icompose(I2, I3))\end{array}$$

PROOF OF THEOREM 5.3:

Equivalence of state sets follows from an identical argument as above. There are many cases to investigate to determine the equivalence of the operations set. The arguments are similar for all of these cases, so we will provide one example. Assnme that the message satisfies

$$m \in (I1.messages \cap I3.messages) - I2.messoges.$$

The operation defined for that m is given by the following set comprehension.

$$\begin{array}{l}operations^{I(12)3}(m) = \\ \{\ s, s' : states^{I(12)3} \\ \mid\ (\ (attribs^{I12} \lhd s, attribs^{I12} \lhd s') \in operations^{I12}(m) \\ \quad \wedge\ (attribs^{I3} \lhd s, attribs^{I3} \lhd s') \in operations^{I3}(m)) \\ \bullet\ \ s \mapsto s'\ \}\end{array}$$

By the definition of $operations^{I12}(m)$, this expands to

$$\begin{array}{l}\{\ s, s' : states^{I123} \\ \mid\ (\ (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\ \quad \wedge\ (attribs^{I2} \lhd attribs^{I1} \lhd s, attribs^{I2} \lhd attribs^{I1} \lhd s') \\ \quad\ \in \text{id}(attribs^{I2} - attribs^{I1} \nrightarrow \mathcal{V}) \\ \quad \wedge\ (attribs^{I3} \lhd s, attribs^{I3} \lhd s') \in operations^{I3}(m)) \\ \bullet\ \ s \mapsto s'\ \}\end{array}$$

We can rewrite the middle clause because the attribute sets are disjoint.

$$
\begin{array}{l}
\{\, s, s' : states^{I123} \\
\mid \; (\quad (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\
\quad \wedge\, (attribs^{I2} \lhd attribs^{I3} \lhd s, attribs^{I2} \lhd attribs^{I3} \lhd s') \\
\quad\quad \in \mathrm{id}(attribs^{I2} - attribs^{I3} \nrightarrow \mathcal{V}) \\
\quad \wedge\, (attribs^{I3} \lhd s, attribs^{I3} \lhd s') \in operations^{I3}(m)) \\
\bullet \;\; s \mapsto s' \,\}
\end{array}
$$

By the definition of $operations^{I23}$, this can be rewritten as follows.

$$
\begin{array}{l}
\{\, s, s' : states^{I12(3)} \\
\mid \; (\quad (attribs^{I1} \lhd s, attribs^{I1} \lhd s') \in operations^{I1}(m) \\
\quad \wedge\, (attribs^{I23} \lhd s, attribs^{I23} \lhd s') \in operations^{I23}(m)) \\
\bullet \;\; s \mapsto s' \,\}
\end{array}
$$

This last set comprehension is equal to $operations^{I1(23)}(m)$, as desired.
$\diamond$ END OF PROOF OF THEOREM 5.3

5.3 External specification

The internal specification implies an ordering on the sequence of state transitions in which an agent can participate. If a designer wants to specify the order of operations to satisfy some constraint, it is not always easy to do by means of the internal specification. In other words, some constraints are not naturally expressible in terms of state transitions. This is a similar conclusion to that reached by Lamport [99], Sufrin and He [158] and Took [162].

Another way to express constraints on the state transitions is to view the constraints as external, meaning that an agent has both an internal and external specification. The agent participates in events and the external constraints are expressible as predicates on event participation. *Process algebras* are specifically designed to accommodate such event descriptions. The standard examples of process algebras are Hoare's Communicating Sequential Processes (*CSP*) notation [81, 82], developed at Oxford, and Milner's Calculus for Communicating Systems (*CCS*) developed at Edinburgh.

We adopt the traces model of a CSP process as the basis for the external specification as a nonempty prefix-closed set of sequences of events.

ExternalSpec

$alphabet : \mathbb{P}\ Event$

$traces : \mathbb{P}\ \mathrm{seq}\ alphabet$

$\langle\rangle \in traces$

prefix_closed $(traces)$

The definition of the predicate prefix_closed is defined in Appendix A.

It would be possible to incorporate more complex process algebra models. We do not ignore the significant variety of process algebra models, most notably the family of algebras derived from Milner's CCS. For the purposes of this thesis, however, a demonstration of the incorporation of a process algebraic technique with the model-oriented axiomatic technique is all that is important, and in doing so we limit ourselves to the simplest case of the CSP traces model. More sophisticated models would allow expression of more sophisticated external constraints. Within the traces model, only *safety* constraints may be expressed, which would allow explicit expressions to rule out undesirable behaviour. *Liveness* constraints which make explicit that desirable behaviour will happen, require a model such as the failures-divergence model of CSP [28]. Some timing constraints may require even more sophisticated process algebras that incorporate a model of time and concurrency beyond the interleaving semantics of the traces or failures-divergence models. Examples in the CSP family are the various timed models of Reed [132] and Davies and Schneider [46, 144, 45]. A timed version of CCS has been provided by Moller and Tofts [111].

Combining external specifications

One of the main features of a process algebra is its compositionality. We will discuss in a later section how that compositionality is achieved in isolated descriptions of external specifications. Then, we will provide both a constructive trace notation, which is described in detail in Appendix C and a more implicit descriptive techniqne of a predicate calculus on traces. In this section, we will concentrate on how to combine external specifications at the agent level, i.e., when the external specification is linked with an internal and commnnication specification. In this case we have two desired combinations—independent combination by interleaving and dependent combination by synchronization.

Interleaved combination is used to allow the behaviours of the separate external specifications to occur in the combiued specificatiou. The traces resulting are all iuterleavings of pairs of traces from the two separate specifications. The description of this interleaving is exactly the same as the interleaving of processes in CSP [82], which we also further define in Appendix C.

$$
\begin{array}{|l}
\textit{InterleaveExt} \\
\hline
\textit{ExternalSpec}^I \\
\textit{ExternalSpec}^J \\
\textit{ExternalSpec}' \\
\hline
\textit{alphabet}' = \textit{alphabet}^I \cup \textit{alphabet}^J \\
\textit{traces}' = \{\ s : \operatorname{seq} \textit{alphabet}' \\
\qquad \mid \ \exists\, t : \textit{traces}^I;\ t' : \textit{traces}^J \bullet s \ \textit{interleaves}\ (t, t') \\
\qquad \bullet \ s\ \} \\
\hline
\end{array}
$$

We can represent this combination as a binary operation, $Ecompose_{int}$.

$$
\begin{array}{|l}
Ecompose_{int} : (\textit{ExternalSpec} \times \textit{ExternalSpec}) \rightarrow \textit{ExternalSpec} \\
\hline
Ecompose_{int} = \{\ \textit{InterleaveExt} \\
\qquad \bullet \ (\theta \textit{ExternalSpec}^I, \theta \textit{ExternalSpec}^J) \mapsto \textit{ExternalSpec}'\ \} \\
\hline
\end{array}
$$

Synchronous combination, as its name suggests, allows for the synchronizing of events between two external specifications. Its definition is similar to that for synchronous parallel combination in CSP, which we have also further defined in Appendix C.

$$
\begin{array}{|l}
\textit{SynchronizeExt} \\
\hline
\textit{ExternalSpec}^I \\
\textit{ExternalSpec}^J \\
\textit{ExternalSpec}' \\
\hline
\textit{alphabet}' = \textit{alphabet}^I \cup \textit{alphabet}^J \\
\textit{traces}' = \{\ t : \operatorname{seq} \textit{alphabet}' \\
\qquad \mid \ (\ t \overset{\text{seq}}{\rhd} \textit{alphabet}^I \in \textit{traces}^I \\
\qquad\quad \wedge\ t \overset{\text{seq}}{\rhd} \textit{alphabet}^J \in \textit{traces}^J) \\
\qquad \bullet \ t\ \} \\
\hline
\end{array}
$$

Sequence range restriction, $\overset{\text{seq}}{\rhd}$, is defined in Appendix A. It operates much like a filter in a functional programming language. We can represent this combination as a binary operation, $Ecompose_{sync}$.

$$
\begin{array}{|l}
Ecompose_{sync} : (\textit{ExternalSpec} \times \textit{ExternalSpec}) \rightarrow \textit{ExternalSpec} \\
\hline
Ecompose_{sync} = \{\ \textit{SynchronizeExt} \\
\qquad \bullet \ (\theta \textit{ExternalSpec}^I, \theta \textit{ExternalSpec}^J) \mapsto \textit{ExternalSpec}'\ \} \\
\hline
\end{array}
$$

It is well known that both of these operators are associative [82, pp. 70,120].

5.4 Communication

The internal specification induces an ordering on the state transitions of the agent and this ordering can be represented by the message labels associated to the transitions via the function *operation*. The external specification gives directly the ordering on events that can occur during *traces* of the behaviour. In the communication specification for an agent, we link the messages of the internal specification to the events of the external specification. Events are communications of messages along channels.

To accommodate this within the agent model, we will introduce explicit channels that are meant as point-to-point, one-way, synchronous communication paths between agents, as in the programming language occam [86, 91]. Though this is a fairly restrictive means of communication, it has been noted that this restriction makes possible tractable algebraic techniques for reasoning rigorously about an occam program [25, 139, 140], which we can then incorporate into the agent development method. We assume a set of all possible channel identifiers.

$$[ChannelID]$$

An event is a message communicated along some channel. We can represent the refined *Event* type as a schema which indicates the channel name and the message.

$$Event == ChannelID \times Message$$

We provide some shorthand notation for accessing the channel and message content of an event.

$$\begin{array}{|l} \mathrm{chan} : Event \rightarrow ChannelID \\ \mathrm{mess} : Event \rightarrow Message \\ \hline \mathrm{chan} = fst \\ \mathrm{mess} = snd \end{array}$$

A channel is associated to a set of message which can be communicated across it. There are four kinds of channels possible. A channel can be used for the input of messages or the output of messages. The combination of an agent which uses a channel for input and an agent which uses the same channel for output creates a synchronized channel. Finally, there is a completely internal channel which handles internal communication within an agent.

We define a function *events* which yields the set of all possible events that can occur along a channel.

$$\begin{array}{|l} \mathrm{events} : (ChannelID \times \mathbf{P}\, Message) \rightarrow \mathbf{P}\, Event \\ \hline \mathrm{events}(c, MS) = \{\ m : MS \bullet (c, m)\ \} \end{array}$$

The communication specification gives all input, output and synchronized channels. These channels are all distinct. Each agent also has a special channel for internal events, which is called τ. From the channel information, we can derive the alphabet of events in which the agent can participate and the set of messages for which operations must be defined. These two derived sets will match the corresponding sets in the external and internal specifications.

Communication

$$
\begin{array}{l}
inputs, outputs, syncs : ChannelID \pfun \mathbb{P}\, Message \\
internal : \mathbb{P}\, Message \\
alphabet : \mathbb{P}\, Event \\
messages : \mathbb{P}\, Message \\
\hline
\text{disjoint}\ (\text{dom}\ inputs, \text{dom}\ outputs, \text{dom}\ syncs, \{\tau\}) \\
alphabet = \text{events}(\!|inputs|\!) \cup \text{events}(\!|outputs|\!) \cup \\
\qquad\qquad \text{events}(\!|syncs|\!) \cup \text{events}(\tau, internal) \\
messages = \text{mess}(\!|alphabet|\!)
\end{array}
$$

Combining communication specifications

Two communication specifications are *channel compatible* if no synchronized channel of one is either an input, output or synchronized channel of the other.

ChannelCompatible

$$
\begin{array}{l}
Communication^I \\
Communication^J \\
\hline
\text{dom}\ syncs^J \cap (\text{dom}\ inputs^I \cup \text{dom}\ outputs^I \cup \text{dom}\ syncs^I) = \varnothing \\
\text{dom}\ syncs^I \cap (\text{dom}\ inputs^J \cup \text{dom}\ outputs^J \cup \text{dom}\ syncs^J) = \varnothing
\end{array}
$$

Two communication specifications are *unlinked* if they are channel compatible and no input channel of one is an output channel of the other.

Unlinked

$$
\begin{array}{l}
ChannelCompatible \\
\hline
(\text{dom}\ inputs^I \cap \text{dom}\ outputs^J) = \varnothing \\
(\text{dom}\ inputs^J \cap \text{dom}\ outputs^I) = \varnothing
\end{array}
$$

Two communicatiou specifications which are unlinked can be combined to yield a third communication specification. Unlinked communication specifications can

share the same input channels or the same output channels. The combined specification forms the union of the message sets associated to each channel identifier. This combination of communication specifications will be used to form the interleaved composition of agents.

InterleaveComm

$$Unlinked$$
$$Communication'$$

$$internal' = internal^I \cup internal^J$$
$$syncs' = syncs^I \cup syncs^J$$
$$inputs' = \{\ c : \mathrm{dom}\ inputs^I \cup \mathrm{dom}\ inputs^J \bullet c \mapsto (inputs^I(c) \cup inputs^J(c))\ \}$$
$$outputs' = \{\ c : \mathrm{dom}\ outputs^I \cup \mathrm{dom}\ outputs^J \bullet c \mapsto (outputs^I(c) \cup outputs^J(c))\ \}$$

We can represent this combination as the binary operation, $Ccompose_{int}$.

$$Ccompose_{int} : (Communication \times Communication) \rightarrow Communication$$

$$Ccompose_{int} = \{\ InterleaveComm \bullet (\theta Communication^I, \theta Communication^J) \mapsto Communication'\ \}$$

Two communication specifications which are channel compatible and which have no common input channels nor common output channels are said to be *linkable* and they can be synchronized. The common input/output channel pairs are made synchronous channels. The messages which are allowed on the synchronized channel are ones which are allowed as input by one agent or output by the other.

Linkable

$$ChannelCompatible$$

$$(\mathrm{dom}\ inputs^I \cap \mathrm{dom}\ inputs^J) = \emptyset$$
$$(\mathrm{dom}\ outputs^I \cap \mathrm{dom}\ outputs^J) = \emptyset$$

$$
\begin{array}{|l}
\textit{SynchronizeComm} \\
\textit{Linkable} \\
\textit{Communication}' \\
\hline
\textit{internal}' = \textit{internal}^J \cup \textit{internal}' \\
\textit{syncs}' = \quad \textit{syncs}^I \cup \textit{syncs}^J \\
\qquad \cup \{\ c : (\mathrm{dom}\ \textit{inputs}^I \cap \mathrm{dom}\ \textit{outputs}^J) \\
\qquad\quad \bullet\ c \mapsto (\textit{inputs}^I(c) \cup \textit{outputs}^J(c))\ \} \\
\qquad \cup \{\ c : (\mathrm{dom}\ \textit{inputs}^J \cap \mathrm{dom}\ \textit{outputs}^I) \\
\qquad\quad \bullet\ c \mapsto (\textit{inputs}^J(c) \cup \textit{outputs}^I(c))\ \} \\
\textit{inputs}' = \mathrm{dom}\ \textit{syncs}' \lhd (\textit{inputs}^I \cup \textit{inputs}') \\
\textit{outputs}' = \mathrm{dom}\ \textit{syncs}' \lhd (\textit{outputs}^I \cup \textit{outputs}') \\
\end{array}
$$

We can represent the synchronized composition of communication specifications as the binary operation, $Ccompose_{sync}$.

$$
\begin{array}{|l}
Ccompose_{sync} : (\textit{Communication} \times \textit{Communicotion}) \nrightarrow \textit{Communication} \\
\hline
Ccompose_{sync} = \\
\{\ \textit{SynchronizeComm} \\
\bullet\ (\theta \textit{Communication}^I, \theta \textit{Communication}^J) \mapsto \theta \textit{Communication}'\ \}
\end{array}
$$

By arguments similar to those given to show that internal composition is associative, we can also show that communication composition operators are also associative.

5.5 Overall Combination

Having specified the componeuts of the refined agent model and bow they are individually composable, we cau now define two operations to compose agents. The two composing functions will represent an interleaving composition, in which the two agents act as one agent by interleaving their individual behaviour, and a synchronizing composition, in which communication will be synchronized between common channels.

Interleaving composition is specified in terms of the general internal specification composition and the interleaved composition of tbe external and communication specifications. We include the constraint on message compatibility defined earlier so that the conditions of Theorem 5.2 are upheld and the resulting composition operator, $compose_{int}$, is associative.

InterleaveCompose

$Agent^I$
$Agent^J$
$Agent'$
MessageCompatible
CombineInt
InterleaveExt
InterleaveComm

The binary operation for interleave composition is $compose_{int}$

$$compose_{int} : (Agent \times Agent) \nrightarrow Agent$$

$$compose_{int} =$$
$$\{\ InterleaveCompose$$
$$\bullet\ (\theta Agent^I, \theta Agent^J) \mapsto \theta Agent'\ \}$$

Arguments in the preceding three sections lead to the proof of the associativity of $compose_{int}$.

Theorem 5.4 *Interleaving composition, when defined, is associative, i.e.,*

$$compose_{int}(compose_{int}(A1, A2), A3) = compose_{int}(A1, compose_{int}(A2.A3)).$$

Synchronizing composition is specified in terms of the general internal specification composition and the synchronized composition of the external and communication specifications. We add the constraint that the attribute sets of the agents to be composed must be disjoint. This is to ensure that the conditions of Theorem 5.3 are upheld in order that $compose_{sync}$ be associative.

SynchronizeCompose

$Agent^I$
$Agent^J$
$Agent'$
CombineInt
SynchronizeExt
SynchronizeComm

$$attribs^I \cap attribs^J = \emptyset$$

The binary operation for interleave composition is $compose_{sync}$

$$compose_{sync} : (Agent \times Agent) \nrightarrow Agent$$

$$compose_{sync} =$$
$$\{\ SynchronizeCompose$$
$$\bullet\ (\theta Agent^I, \theta Agent^J) \mapsto \theta Agent'\ \}$$

Arguments in the preceding three sections lead to the proof of the associativity of $compose_{sync}$.

Theorem 5.5 *Synchronous composition, when defined, is associative, i.e.,*

$$compose_{sync}(compose_{sync}(A1, A2), A3) =$$
$$compose_{sync}(A1, compose_{sync}(A2, A3)).$$

Since these composition operators are associative, we can define operations which compose arbitrary nonempty sequences of agents by means of the folding operations of standard functional programming [24]. Refer to Appendix A for definitions of the folding operations in standard Z.

$$composeall_{int},$$
$$composeall_{sync} : \mathrm{seq}_1\ Agent \nrightarrow Agent$$

$$composeall_{int} = foldl1\ compose_{int}$$

$$composeall_{sync} = foldl1\ compose_{sync}$$

The composition operators are not mutually associative, but in practice this will not matter. Interleaving is used to build up a specification of an agent that will have a private state. Synchronization is used to connect the separate agents, i.e., ones which have a private state.

5.6 Interpretations of agents

The communication specification now provides the link necessary for combining an internal specification and an external specification. What remains is to describe the refinements of the internal and external interpretation relations.

5.6.1 Internal interpretation

We want to derive the internal interpretation relation I_A^{int} for an agent which maps traces of events that an agent can participate in to the possible states that the agent can be in after participating in the trace. Following the approach of Chapter 2, we first extend the *operations* mapping from the internal specification. However, since

that mapping uses messages to index the state transitions and not events, we must also extract the message content of the event.

$$\begin{array}{|l} opExtend\_ : Agent \rightarrow \mathrm{seq}\, Event \rightarrow (State \leftrightarrow State) \\ \hline opExtend_A(trace) = A.inits \lhd (\fatsemi/(trace \fatsemi \mathrm{mess} \fatsemi A.operations)) \end{array}$$

A difference to note from the development of Chapter 2 is that every event has an associated state transition, instead of jnst input events. We refer to sequences of events in this definition as traces instead of programs to alert the reader of the difference. It is the role of an event which distinguishes stimuli (input) from response (output). The domain of *opExtend* manifests the constraints on event sequencing which arise from the internal specification. The external specification is already expressed in terms of event sequences. I_A^{int} mnst be consistent with both.

$$\begin{array}{|l} I\_^{int} : Agent \rightarrow (\mathrm{seq}\, Event \leftrightarrow State) \\ \hline (trace, s) \in I_A^{int} \Leftrightarrow (\quad \exists s_0 : A.inits \bullet (trace, (s_0, s)) \in opExtend_A \\ \qquad\qquad \wedge\ trace \in A.traces) \end{array}$$

It is important to note in this definition, that the overall interpretation is given as a conjunction of internal and external constraints. This was the main motivation for the hybrid notation of the Sufrin and He model. In that formal model, however, the external description was defined to constrain the transitions possible from the internal description, but not vice versa. However, it was clearly the *intent* of those authors to allow the iuternal specification to constrain the traces possible from the external description. We mention this point as another example of how a formal approach can uncover inconsistencies between informal requirements (the intent mentioned above) and the deliverable (tbe actual formalism).

The *histories* of an agent, written $\mathcal{H}[\![\ A\ ]\!]$, give a recording of the events in wbich it can participate and are derived from the domain of I_A^{int}.

$$\begin{array}{|l} \mathcal{H}[\![\ \_\ ]\!] : Agent \rightarrow \mathbb{P}(\mathrm{seq}\, Event) \\ \hline \mathcal{H}[\![\ A\ ]\!] = \mathrm{dom}\, I_A^{int} \end{array}$$

There are three properties which must be satisfied by the internal interpretation relation and its derivatives:

- the set of histories of an agent is prefix closed, so that for an agent to participate in a sequence of events it must bave participated in each prefix;
- the internal interpretation relation respects the history of interaction, so that any state the agent can attain must have been reached from a previously attainable state; and

- the internal interpretation relation is nonempty, so that every agent has some behaviour.

We will prove each of these properties of the internal interpretation relation in turn as theorems.

Theorem 5.6 *The set of traces of an agent is prefix closed, i.e.,*

$$\forall A : Agent \bullet \text{prefix\_closed } \mathcal{H}[\![A]\!].$$

PROOF OF THEOREM 5.6:

To prove this theorem, we need Lemma 5.1, which states that *opExtendA* obeys a nice compositional property.

Lemma 5.1

$$\begin{array}{l} \forall A : Agent;\ s : A.states;\ s_0 : A.inits;\ hd, tl : \text{seq } Event \\ \bullet\ (hd \frown tl, (s_0, s)) \in opExtend_A \Leftrightarrow \\ \qquad \exists s' : A.states \\ \qquad\quad \bullet\ (\ \ (hd, (s_0, s')) \in opExtend_A \\ \qquad\qquad \wedge\ (tl, (s', s)) \in (tl \fatsemi \text{mess} \fatsemi A.operations)) \end{array}$$

PROOF OF LEMMA 5.1:

Because of layout considerations, in the following argument we have substituted the expression *Aops* for the correct expression *A.operations*.

$$\begin{array}{ll} (hd \frown tl, (s_0, s)) \in opExtend_A & \text{[assumption]} \\ \Leftrightarrow (s_0, s) \in \fatsemi/(hd \frown tl \fatsemi \text{mess} \fatsemi Aops) & \text{[defn. of } opExtend_A, s_0 \in A.inits] \\ \Leftrightarrow (s_0, s) \in \fatsemi/\left((hd \fatsemi \text{mess} \fatsemi Aops) \frown (tl \fatsemi \text{mess} \fatsemi Aops)\right) & [\fatsemi \text{ dist. over } \frown] \\ \Leftrightarrow (s_0, s) \in (\fatsemi/(hd \fatsemi \text{mess} \fatsemi Aops)) \fatsemi (\fatsemi/(tl \fatsemi \text{mess} \fatsemi Aops)) & \text{[defn. of } \fatsemi/] \\ \Leftrightarrow \exists s' : A.states & \text{[defn. of } \fatsemi] \\ \quad \bullet\ (\ \ (s_0, s') \in (\fatsemi/(hd \fatsemi \text{mess} \fatsemi Aops)) & \\ \qquad \wedge\ (s', s) \in (\fatsemi/(tl \fatsemi \text{mess} \fatsemi Aops))) & \\ \Leftrightarrow \exists s' : A.states & \text{[defn. of } opExtend] \\ \quad \bullet\ (\ \ (hd, (s_0, s')) \in opExtend_A & \\ \qquad \wedge\ (tl, (s', s)) \in (tl \fatsemi \text{mess} \fatsemi Aops)) & \end{array}$$

◇ END OF PROOF OF LEMMA 5.1

Continuing with the proof of Theorem 5.6, let $tr \in \mathcal{H}[\![A]\!]$ and $hd \frown tr$, i.e., $hd \frown tl = tr$. We need to show that $hd \in \mathcal{H}[\![A]\!]$.

$$
\begin{array}{ll}
hd \frown tl \in \mathcal{H}[\![A]\!] & \text{[assumption]} \\
\Leftrightarrow hd \frown tl \in \operatorname{dom} I_A^{int} & \text{[defn. of } \mathcal{H}[\![A]\!]\text{]} \\
\Leftrightarrow \exists s : A.states;\ s_0 : A.inits & \text{[defn. of } I_A^{int}\text{]} \\
\quad \bullet\ (\quad (hd \frown tl, (s_0, s)) \in opExtend_A & \\
\qquad \wedge\ hd \frown tl \in A.traces) & \\
\Leftrightarrow \exists s' : A.states;\ s_0 : A.inits & [\ \text{Lemma 5.1} \\
\quad \bullet\ (\quad (hd, (s_0, s')) \in opExtend_A & \text{prefix\_closed } (A.traces)\ ] \\
\qquad \wedge\ hd \in A.traces & \\
\Leftrightarrow hd \in \operatorname{dom} I_A^{int} & \text{[defn. of } I_A^{int}\text{]} \\
\Leftrightarrow hd \in \mathcal{H}[\![A]\!] & \text{[defn. of } \mathcal{H}[\![A]\!]\text{]}
\end{array}
$$

◇ END OF PROOF OF THEOREM 5.6

Theorem 5.7 *The internal behaviour of an agent is history-closed, i.e.,*

$$
\begin{array}{l}
\forall A : Agent;\ (tr, s) : I_A^{int} \\
\quad \bullet\ \exists s' : A.states;\ tr', tr'' : \operatorname{seq} Message \\
\qquad |\quad tr' \frown tr'' = tr \\
\qquad \bullet\ (\quad (tr', s') \in I_A^{int} \\
\qquad\quad \wedge\ (s', s) \in (tr'' \fatsemi \operatorname{mess} \fatsemi A.operations)
\end{array}
$$

PROOF OF THEOREM 5.7:

Assnme we have an agent A with $(tr, s) \in I_A^{int}$ and $tr' \frown tr'' = tr$. If $tr' = tr$, then we can satisfy the conditions of the theorem by letting $s' = s$ and $tr'' = \langle\ \rangle$. So assume that $tr' \neq tr$. By Lemma 5.1, we know that

$$
\begin{array}{l}
\exists s' : A.states;\ s_0 : A.inits \\
\quad \bullet\ (\quad (tr', (s_0, s')) \in opExtend_A \\
\qquad \wedge\ (tr'', (s', s)) \in opExtend_A)
\end{array}
$$

Since $tr \in A.traces$ and $A.traces$ is prefix closed by definition, we conclude that $(tr', s') \in I_A^{int}$, as desired.
◇ END OF PROOF OF THEOREM 5.7

Theorem 5.8 *For any agent A, the internal interpretation relation is nonempty.*

PROOF OF THEOREM 5.8:

Since $A.inits$ is nonempty, this theorem is satisfied by demonstrating that, for all agents A and initial state $s_0 : A.inits$, the element $(\langle\,\rangle, s_0)$ is in I_A^{int}. First, we show that $opExtend_A\ \langle\,\rangle = \mathrm{id}\, State$.

$$
\begin{aligned}
&opExtend_A\ \langle\,\rangle \\
&\quad = \fatsemi/(\langle\,\rangle_{\mathrm{seq}\, Message} \fatsemi \mathrm{mess} \fatsemi A.operations) && \text{[defn. of } I\text{]} \\
&\quad = \fatsemi/(\langle\,\rangle_{State \leftrightarrow State}) && \text{[defn. of } \fatsemi\text{]} \\
&\quad = \mathrm{id}\, State && \text{[defn. of } \fatsemi/\text{]}
\end{aligned}
$$

Since $(s_0, s_0) \in \mathrm{id}\, State$, we have satisfied the first predicate in the definition of I_A^{int} because

$$(\langle\,\rangle, (s_0, s_0)) \in opExtend_A.$$

Finally, $\langle\,\rangle \in A.traces$ by the definition of *ExternalSpec*. Therefore, the internal interpretation relation of an agent is never empty, since it contains $(\langle\,\rangle, s_0)$.
◇ END OF PROOF OF THEOREM 5.8

5.6.2 External interpretation

In Chapter 2, we also defined an external interpretation relation, $I_{\_}^{ext}$ to reflect the overall stimulus-response behaviour of an agent. The importance of this interpretation relation was twofold. It provided an external specification of an agent, so that the translations of the interaction framework could be viewed as specifications of agents themselves. It also was used to define some interactive properties, such as predictability, honesty and trustworthiness.

External specification is now explicit in the agent model, but it includes more information than the overall stimulus-response and it also includes event behaviour that could be excluded by the internal specification. I_A^{ext} is defined on the traces of A, applying only to legal behaviours of the agent, and it filters out the input and output events. The stimuli of an agent are the events that can occur along the input channels. The responses are the events that can occur aloug the output channels.

$$
\begin{array}{|l}
stimuli, responses : Agent \to \mathbb{P}\, Event \\
I_{\_}^{ext} : Agent \to (\mathrm{seq}\, Event \leftrightarrow \mathrm{seq}\, Event) \\
\hline
stimuli(A) = \mathrm{events}(\!|A.inputs|\!) \\
responses(A) = \mathrm{events}(\!|A.outputs|\!) \\
I_A^{ext} = \{\ t : \mathcal{H}[\![\ A\ ]\!] \\
\qquad \bullet\ t \overset{\mathrm{seq}}{\rhd} stimuli(A) \mapsto t \overset{\mathrm{seq}}{\rhd} responses(A)\ \}
\end{array}
$$

Because $\mathcal{H}[\![A]\!]$ is prefix closed, we can show that both the domain and range of I_A^{rrt} are prefix closed as well.

5.7 Conclusions

In this chapter, we have refined the model of the agent. The specification of an agent has been split into three parts to define the internal state-based behaviour, the external event-based behaviour and the communication specification which links the internal and external specifications. We have described two composition operators on agents. The first corresponds to the interleaved composition of dependent agents which will aid in the gradual development of complex agents sharing attributes and messages. The second composition is a synchronization of independent agents which share no attributes and communicate via synchronized message passing.

The agent model presented is good for theoretical use. We can express and prove general interactive properties of agents using this model. In Chapter 7 we will use the refined model as a basis for the reformulation of the interactive properties expressed in Chapter 4 and introduce some new ones. The agent model, however, is not a very good design notation. We will justify this claim in the next chapter and present a language for the description of agents which can map back into the agent model.

Chapter 6

A language for describing agents

Up to this point in the thesis, the only formal notation we have used has been Z. We believe that strict adherence to Z as a design notation for agents is not desirable. The way a designer conceptualizes an agent's behaviour must be more directly captured in the agent language than standard Z allows. We will, therefore, provide a new language for the description of agents. We want the new language to be flexible. The properties we express on an agent relate the events and the state, so it is important that we can describe the event-state behaviour. As described in the last chapter, sometimes it is easier to describe such a desired behaviour via the internal state description linked with a communication description and sometimes it is easier to do directly via the external event description.

The agent language is a formal notation, and there are many existing formal notations which are increasingly being used in both academic and industrial circles. We want to take advantage of the familiarity with those existing notations. However, most of the notations are either more suited to the state-based description or the event-based notation, and so they do not alone satisfy the expressiveness requirement described above. Following on from Sufrin and He's model of interactive processes [158], we propose a hybrid notation which marries a model-oriented descriptive technique for the internal description and a process algebraic technique for the external description. Elsewhere, we have described a version of the agent language which uses an algebraic notation for the internal description [69], but the semantics for such a language we have not defined.

Overview of chapter

In Section 6.1, we justify the need for an agent notation different from standard Z and we give an overview of some other agent-like notations. In Section 6.2, we provide a language for describing agents and we outline the mapping from that syntactic domain to the semantic model, further details of which can be found in Appendix C. The best way to explain how the agent language is used is by example,

and so we offer several examples in Section 6.3. We have placed a heavy emphasis on familiarity with the new notation, which has resulted in a hybrid language resembling established model-oriented langnages, such as Z or VDM, and process algebra notations, such as CSP or CCS. Any language exerts an influence on its users—some things will be easy to express within the language and some things will be difficult. Such an influeuce is not a bad thing, as long as we are aware of it and we recognize the limitations it implies. At the end of this chapter we will summarize the limitations of the agent language.

6.1 Notations for agents

Although the model presented in Chapter 5 will be adequate for deriving the interactive properties to be discussed in Chapter 7, it does not satisfy the requirement of natural expressiveuess. Before we present the agent language, we explore in this section some other possible languages for our agents and other object-oriented formalisms.

6.1.1 The standard Z notation

We have used the Z notation to present all of the formalisms so far in this thesis, and we will continue to use Z as the means for mathematical expression and reasoning within the agent model. However, we do not favour Z as the design notation for agents for two main reasons.

Our first criticism of Z is the lack of modularity it provides in standard use. Whereas in most tutorials on the Z language a modular development approach is advocated, any modularity in resulting specifications is left to the reader to extract. The principal features of an internal specification of an agent—the state space, the initial states, and the operations on the state space—are separately describable in Z. When we want to bundle these components together, it is possible in Z, but not natural to the standard Z development. The possibility was demonstrated by Sufrin and He [158], so to justify our argument, we will provide an example of the specification of a window in a multi-windowing environment using their model of a process, given below.

$[E]$

SandHProcess[*S*]

$\alpha : \mathbf{P}\, E$

$\hat{\beta} : E \nrightarrow (S \leftrightarrow S)$

$Trace : \mathbf{P}(\mathrm{seq}\, E)$

$\iota : \mathbf{P}\, S$

$\alpha = \mathrm{dom}\, \hat{\beta}$

$\forall s, t : \mathrm{seq}\, E \bullet s \frown t \in Trace \Rightarrow s \in Trace$

For our purposes, we can equate the type E with the type *Message*, and S with *State*.

We will begin by giving a Z description of the functionality of the individual window. A window has two representations, depending on whether it is open or closed. When closed it takes the form of an icon, which is one from a set of all possible icons, denoted *ICON*.

$[ICON]$

When open, the window covers a rectangular region on a finite coordinate plane representing the visual display. This plane is commonly viewed as a pixel plane with boundaries in the horizontal (x-axis) and vertical (y-axis) directions. The type *PIXEL* will represent points in this finite coordinate plane.

$xmax, ymax : \mathbf{N}$

$PIXEL == 0 \,..\, xmax \times 0 \,..\, ymax$

For our purposes, a window is completely defined when we have the following information on it:

- the icon to represent it when closed and its position in *PIXEL* space
- the position and extent of its rectangular region in *PIXEL* space when open
- an indication of whether the window is open or closed

The schema type *WindowState* describes the state of such a window. Each aspect mentioned above is represented directly by a component of the schema type *WindowState*.

WindowState

$icon : ICON$

$iconpos, winpos, winsize : PIXEL$

$status : open \mid closed$

We could add some constraints on the window state. For example, we could constrain the size of the window such that the whole window is contained in *PIXEL* space. For simplicity, we ignore such constraints for the moment.

When a window is created, we can stipulate that it satisfies certain constraints beyond any given as the state invariant of *WindowState* (had we given any). Initialization constraints are by convention detailed by the schema *WindowInit*, and the purpose of this schema is to give the subset of all possible states in which a window can be initially. This schema only contains a copy of the state after the initialization (the window is assumed not to exist before initialization). In our example, we will stipulate that a window begins with status *open*.

WindowInit

$$WindowState'$$

$$status' = open$$

Some of the normal operations performed upon a window would be to open or close it, to move it (when open or closed) or to resize it when open. We indicate a window operation by a schema description containing the declaration $\Delta WindowState$ which contains two copies of window schema binding to represent the window before (using undashed component names) and after (nsing dashed component names) the operation. We can specify the operation to open a window by requiring that the status of the window indicate that it is closed before the operation and open afterwards. No other componeut of the window is changed. [1]

OpenWindowOp

$$\Delta WindowState$$
$$\Xi(WindowState \setminus \{status\})$$

$$status = closed$$

$$status' = open$$

This operation definition is equivalent to the following expanded schema.

[1] Note the use of the Ξ convention along with schema hiding ($\setminus$) to give such framing conditions. The hidden components are precisely those we wish the operation to be able to change; the other remain the same. Though not widely publicized, this technique of explicitly naming the framing conditions of operations is good practice [106, 27].

OpenWindowOp

$$WindowState$$
$$WindowState'$$

$$status = closed$$
$$status' = open$$
$$icon' = icon$$
$$iconpos' = iconpos$$
$$winpos' = winpos$$
$$winsize' = winsize$$

Similar descriptions can describe other operations on a window, such as closing, moving or resizing. As an example of an operation which takes an argument, we specify how a window is moved. Either the open window or iconified window position can be altered, so we define the general move operation in two parts to cover the two different cases. The overall window repositioning operation is then the disjunction of those two separate operations. When the window is closed, the argument to the repositioning operation indicates the new position for the icon.

MoveWindowClosed

$$\Delta WindowState$$
$$\Xi(WindowState \setminus \{iconpos\})$$
$$newpos? : PIXEL$$

$$status = closed$$
$$iconpos' = newpos?$$

When the window is open, the argument indicates the new position for the rectangular region.

MoveWindowOpen

$$\Delta WindowState$$
$$\Xi(WindowState \setminus \{winpos\})$$
$$newpos? : PIXEL$$

$$status = open$$
$$winpos' = newpos?$$

$$MoveWindowOp \mathrel{\widehat=} MoveWindowClosed \lor MoveWindowOpen$$

The window is represeuted as a process in the following manner. The state set is given by *WindowState* and the initial states (ι) are characterized by *Windowinit*. The message set contains elements that can be linked to the schema operations realized as relations on *WindowState*.

$$
\begin{array}{l}
E ::= \mathbf{open} \\
\quad \mid \mathbf{close} \\
\quad \mid \mathbf{move}(\!\langle PIXEL \rangle\!) \\
\quad \mid \mathbf{resize}(\!\langle PIXEL \rangle\!) \\
\quad \vdots
\end{array}
$$

$$
\begin{array}{|l}
\hline
Window : SandHProcess[WindowState] \\
\hline
Window.\alpha = E \\
Window.\iota = \{\ WindowInit \bullet \theta Window'\ \} \\
Window.\hat{\beta}(\mathbf{open}) = \\
\quad \{\ OpenWindowOp \bullet \theta WindowState \mapsto \theta WindowState'\ \} \\
\forall p : PIXEL \\
\bullet\ \ Window.\hat{\beta}(\mathbf{move}(p)) = \\
\qquad \{\ MoveWindowOp \\
\qquad \mid\ newpos? = p \\
\qquad \bullet\ \theta WindowState \mapsto \theta WindowState'\ \} \\
| \\
\quad \vdots \\
\hline
\end{array}
$$

Sufrin and He [158], showed that it is even possible to give the external specification within Z, and we offer further proof of that in Appendix C with a full semantics for a constructive trace language based on CSP. So there is no feature of a process (or agent) which cannot in principle be represented in Z. But this representation is achieved through rather roundahout measures, none of which is difficnlt, but all of which seem unnecessary to the description of an agent.

Our second criticism of Z as the agent notation again arises because Z is not specifically geared to describe functioning entities (like an agent or process) in isolation within a system of other entities. Z has a limited ability to express communication between schemas, sequential composition and piping operators being defined in the schema calculus having only the limited possibility of communicating with one other schema. The stimulus-response model dictates that the agents be able to communicate to an arbitrary number of other agents in response to any stimulus received. Though extensions to Z which we discnss below remedy the earlier problem about bundling internal information into one object, no previous formalism adequately addresses this failing.

6.1.2 Object-oriented notations and Z

Our notion of an agent is somewhat similar to that of an object in object-oriented programming parlance. With the advent of object oriented programming languages, there has been a change in the way many designers view the systems that they build [43]. A system can now be naturally viewed as a collection of objects which pass messages that cause changes in their neighbours, that is, designers have adopted a stimulus-response view of their systems. In response to the increasing acceptance of object-oriented programming notations in industry, researchers have attempted to provide a formal notation to represent the largely informal and intuitively appealing concepts of objects.

The roots of object-oriented programming can be traced to data abstraction [102], in which only the means for transforming an underlying data structure, not the procedure, are made apparent to its user. Though data abstraction within an algebraic framework was initiated at least as far back as 1978 by Guttag and Horning [65] and Goguen *et al.*[60], it has only been investigated more recently within a model-oriented axiomatic approach. In this section we will discuss a few of the techniques distinct from our agent model that have been offered, in a roughly chronological order. These techniques have all influenced the development of the agent model, though none seems to address the issues around communication as well as the agent model.

Promotion

The description of a window and the operations that can be performed on it is easiest when done in isolation, that is, without consideration of any other windows which may coexist. The schema definitions describe the kinds of transformations on windows that we want to be possible. The operation definitions describe transitions on all possible windows; given *any* element of type *WindowState*, the operation definitions provide a description of when and how that element can be transformed to another element in *WindowState*. This is subtly different from an interpretation of what operations on windows as objects in an interactive windowing environment represent. In that case, we create a window and that marks the beginning of its existence as an entity (or object or process or agent). This entity has characteristics (or components or attributes) which fully describe it at any point during its existence and there are operations which can be performed on the entity to change those characteristics *but without changing the identity of the window within the system.*

This last point is very important point because it embodies another criticism of Z, namely that it does not allow for identification of the objects in a system so that changes to one object can be isolated. We do not feel, however, that this criticism is valid, as the standard Z development method provides *promotion* as a

means of objectifying isolated specifications in order to incorporate them directly in a larger global system description. The technique of promotion is common amongst Z practitioners, being first explained by Morgan and Sufrin in their description of the Unix filing system [117], and given a more rigorous mathematical treatment by Woodcock [171].

A fundamental characteristic of an object is its identity. To promote the local window description into the more global window manager level, we need a way of identifying the different instances of windows, all of which are of the same type *WindowState*. We introduce a set of window identifiers, *WINID*, so that the identification of individual windows is modelled as a partial function from identifiers to the schema type *WindowState*. The windows that the manager "knows" about are precisely those in the domain of this function. The window manager also keeps track of the current selected window or windows, to which all future inpnt is directed. The window manager may contain other information as well, but for our present purpose we need not bother with any further detail.

$[WINID]$

WindowManager

$$windows : WINID \pfun WindowState$$
$$known : \mathbb{P}\ WINID$$
$$selected : \mathbb{P}\ WINID$$

$$known = \mathrm{dom}\ windows$$
$$selected \subseteq known$$

We have already defined operations such as *MoveWindowOp* on a single, isolated window because they were more conveniently described in that context, rather than in the context of the window manager. We can promote the local operations via the promotion schema *WindowPromote*.

WindowPromote

$$\Delta WindowManager$$
$$win? : WINID$$
$$\Delta Window$$

$$win? \in known$$
$$\theta Window = windows\ win?$$
$$windows' = windows \oplus \{win? \mapsto \theta Window'\}$$
$$known' = known$$
$$selectcd' = selected$$

This promotion schema will allow us to embed the local window operations as operations of the window manager with minimal changes to their definition. The advantage to this method is that in the definition of the window operations we did not have to worry about properties of the window manager, which makes their definitions not only simpler but more natural to express. For example, when defining the move window operation it should not concern the specifier what other windows are known to the window manager. Operations on windows can be defined directly as operations at the level of the window manager via this promotion schema.

$$WMOpen\,WindowOp \mathrel{\widehat=} (Open\,WindowOp \land WindowPromote) \setminus \Delta WindowState$$

$$WMClose\,WindowOp \mathrel{\widehat=} (Close\,WindowOp \land WindowPromote) \setminus \Delta WindowState$$

$$WMMove\,WindowOp \mathrel{\widehat=} (Move\,WindowOp \land WindowPromote) \setminus \Delta WindowState$$

$$WMResize\,WindowOp \mathrel{\widehat=} (ResizeWindowOp \land WindowPromote) \setminus \Delta WindowState$$

In our example, we did not want the global state to affect the local operations. Though the promotion technique can cope with such interference by the global state by suitable parameterization of the global operations, it is more in line with the spirit of data abstraction to avoid such interference. A limitation on promoted operations is that they only affect that part of the global state which was promoted. It is for this reason that in the operation promotion schema *WindowPromote* that we have explicitly stated that the *known* and *selected* components remain unchanged because these are aspects of the global window manager state that are separate from the individual windows. Some operations on windows are only relevant at the level of the window manager. Examples of such operations are creation and destruction of windows. Whereas the conditions for an initialized window or a terminable window were described in isolation from the window manager, the operation of creating and destroying a window can only be defined at the more global level. Whereas the lone window did not exist before creation and after termination, the window manager existence subsumes that of any window it manages.

The main advantage of promotion is that the effects of operations can be isolated to the smallest part of a complex system. This is of great advantage for describing the functionality of a complex system since the specification takes on a compositional look. The power of promotion for layering the description of a complex system is the major contribution first provided by Morgan and Sufrin. A further advantage of promotion is given by its more formal treatment by Woodcock in which we can see that the modularization allows for a proof management system

to prove the properties of a complex specification. Isolation of proof obligations is also important in our agent language.

Promotion captures the identity of objects, but it does not address very adequately the second point about object-oriented notations, that of conceptualization. As Woodcock points out[171], promotion advocates an "onion skin" approach to system development. The windowing system is viewed in layers of functionality, as shown in Figure 6.1. The innermost layer represents the single window, followed

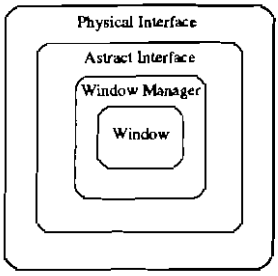

Figure 6.1: The "onion skin" view of a windowing system

by the multi-windowing environment, the abstract interface (in terms of logical input and output devices) and the physical interface. By contrast, an object view of the windowing system may look like Figure 6.2. The onion skin view shows the

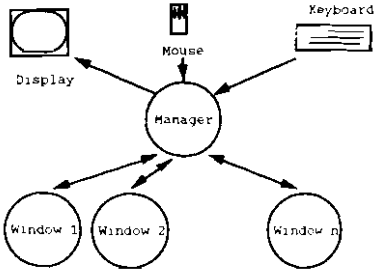

Figure 6.2: The object view of a windowing system

inheritance relationship, whereas the object view shows the instance relationship and communication.

In the next few sections, we will summarize some of the formalisms more in line with the object view.

Schuman and Pitt

Schuman aud Pitt [146], first suggested in 1987 a variant on the Z notation which was specifically designed to meet the ueeds of object-oriented system design and conform to its "pragmatic appeal". Schuman, Pitt and Byers [145] followed three years later with an interpretation of classes in their earlier model as concurrent processes. Though they recognized the need to relate state declaration with initialization and operations more strongly than is done with Z (or VDM), they deliberately did not enforce that relationship syntactically, as we desire to do.

They uote that one of the prime features of a concurrent specification techniques is the ability to reason about the behaviour of the individual processes (or objects) and about combinations of the several processes. They take minimalist approach to process behaviour, characterizing a process in terms of trace/implication pairs. Traces are exactly the same as for our agents above, that is, a record of events in which the process has participated. The events are operation names which are defined by a pre- and postcondition semantics on the components of the state. Implications are that information on the state of a process which can be inferred as a result of the postconditions of the operations in the trace. Communication is in terms of data flow, so synchronization results naturally from conjunction of pre and postconditions. They claim this to represent the most natural form of concurrency, a point which is very debatable. An advantage of our agents is in what we believe to be a more natural facility for expressiug the communication between the separate components.

Schuman, Pitt and Byers provide two means for composing, inheritance aud instantiation. Inheritance occurs when a process C can incorporate and extend the process (or class) information of A and B to define a new process which is a subclass of hoth A and B. Instantiation occurs when C incorporates named versions of A and B in its definition. The mathematical distinction between inheritance and instantiation is that between normal union and disjoint union, and this distinction is similar to the conditions for independent synchronization and interleaving in our agent model, though we do not need only use normal union on attrihute/value mappings to obtain the composition in both cases.

Hall's object-oriented conventions in standard Z

Hall has used standard Z with special conventions designed specifically to deal with object-oriented considerations in software development. The main convention is the nse of special components in the state definition as a means of object identification. This identifier is referred to as *self* in the state definition. We did the same in promotion for the window example hy introducing a set of window identifiers, *WINID*. Hall's convention suggests that this be performed from the start. Operations on the state of an object are not intended to change its identity, and so this can be

made explicit by the addition of the predicate $self' = self$ for all operations.

Hall is also quick to point out the lack of state/operation bundling in Z, but rather than suggest an alternative, he says that the convention is to assume that all operations defined are the only ones possible for the object. This seems reasonable, but it does not address the problem of bundling.

The semantics of objects are again in terms of the traces of operations in which the object can participate. He gives examples of how a mechanical procedure can convert a schema definition of an operation to its relational equivalent. This procedure is identical to the one we showed earlier in converting the window specification to an element in *SandHProcess*, except that Hall uses only the input parameters as the index to the operation. In the agent model this would be equivalent to modelling the operations function as a relation between operations and state transitions instead of a function, and this decision appears very dubious. Hall admits that this mechanical definition cannot he defined in Z, as we have noted, and so he calls upon a convention to perform the mechanical conversion.

The set of conventions advocated by Hall has been used with apparent success in the specification of "a substantial piece" of software written in Objective-C. A very interesting case study was performed using the Model-View paradigm of Smalltalk. We will discuss the formalization of such interactive architectures in Chapter 8.

Object-Z

Research at Queensland has resulted in the development of Object-Z, a true extension to Spivey's semantics for standard Z [34, 55, 56]. Object-Z introduces a class structure whose explicit purpose is to bind together a state description with its invariant, initialization and operations. The semantics of a class is given by the events in which the objects (variable instances) of a class participate. An event is defined as one of the class operations along with the before and after state bindings associated to that instance of the operation application. A trace of such events is called a history.

A relatively minor extension to Spivey's semantics [153] allows for the description of classes. One of the main features of the extension is the addition of roles which are used to distinguish the use of various identifiers as either input or output arguments to an operation definition or before or after values of the state. The *pre* and *post* roles are used to define legal histories of a class instance, since *post* values of the state after one operation must be the *pre* values of the state before the next operation.

Beyond the constraints on state transitions that are defined by the operations, there is also the ability to express constraints on the histories of a class instance explicitly using a temporal logic predicate language. However, this relationship between external specification over histories and the normal internal specification

is not completely formalized in [55]. The authors claim that this will allow the specification of liveness properties, in which assertions can be made about what good properties will occur in the system. Without a clear relationship between the external and internal specifications, such a statement is hard to back up. The external specification may express some liveness criterion, but that behaviour could very well be disallowed by the internal description.

Object-Z provides a simple and very useful extension to Z by the addition of classes. What remains to be done with Object-Z is clear. Just as there is a schema calculus in standard Z which allows for the manipulation and composition of schemas, there needs to be a class calculus in Object-Z which addresses the composition of, and communication between, classes. We feel that our agent model addresses some of those questions.

Whysall and McDermid

Whysall and McDermid have also proposed a means of adding more structure to Z specifications to make them more amenable to object-oriented description via data abstraction [167, 168]. The description of objects is split into two parts, the *export* and the *body* specifications. They make explicit, as we do, that one of the main advantages of such structure is to allow composition of modules, both for the purpose of description of large systems and for the delegation of proof obligations for properties of the large systems. These two specifications roughly correspond to our external and internal interpretation relations on agents.

The export specification gives the input/output trace behaviour of the object. The notation provided allows for this specification in an algebraic language which is based on observational equivalence of method traces. Observational equivalence in this case is similar to external indistinguishability ($\_ \overset{ext}{\equiv} \_$) we defined in Chapter 4. The body specification dictates how the methods behave on the underlying state of the object. This is similar to our internal specification. Whysall and McDermid define the relationship which must hold between an object's export and body specifications, namely that they cannot be inconsistent with each other. Traces of method invocations allowed by the export specification must be allowed by the body specification.

Z++

Work at Oxford has recently resulted in an object-oriented extension to Z called Z++ [100]. A separate notation has been added which performs precisely the bundling of state specification and operations. It has been shown how this notation maps down to more conventional Z. This notation does not consider communication at all between separate objects because it has been developed based on a three-

level design of systems in which each level uses operations of the level below, like the "onion skin" model associated to promotion discussed earlier. Z++ is a *wide spectrum* language because it contains both nonexecutable specification statements and potentially executable procedural statements (in UNIFORM [154]).

6.1.3 Other concurrent notations

We have explicitly incorporated a process algebra model of the external specification of agents because it is a simple compositional model which addresses concurrency. Process algebras gain their simplicity and power from their deliberate disregard of state information. But as Josephs points out [92], this is not always a positive feature in the design of complex systems, as the success of model-oriented specification techniques such as Z and VDM has shown. Both Josephs and He [76, 77] have developed state-based versions of communicating processes in which the transitions on the explicit state space define the traces of behaviour. It was He's work which influenced the hybrid approach of the Sufrin and He model of interactive processes and our own extension of that model in terms of agents.

Notations for concurrency abound; an adequate review of these notations is not within the aims and scope of this thesis. We take time here to mention the Josephs and He models as signposts for the trend in recent years to develop decent models of concurrency with explicit state. Our agent model is such an attempt which has arisen out of the special needs to express interactive properties of a system design. One other attempt of note is the work by Morgan and Woodcock [170] in which a weakest precondition semantics is used to define CSP-like combinators for a concurrent extension to Dijkstra's guarded command language.

6.2 The agent language

In this section, we will describe the language for agents and outline how the notation maps to the model of agents. The agent language is as far as possible a mixture of Z and CSP, so there is no need to give a detailed semantics for the language in the body of this thesis. The interested reader is directed to Appendix C for further details of the semantics for the agent language.

As we have stated, there are two ways to combine agents—by the interleaving of agents which may share common events or attributes, or by the synchronization of agents with disjoint attribute sets. Existing agent definitions can be combined, therefore, by interleaving them or synchronizing them. In addition, we can constructively define an agent by giving the internal, external and communications specifications directly. The examples in Section 6.3 show how and when each method of agent description is used.

An interactive system is a mapping from agent identifiers to the set of agents in *Agent*. We introduce a set of possible agent identifiers.

$$[AgentID]$$

$$IntSys == AgentID \pfun Agent$$

The system semantic function, $\mathcal{S}[\![\_]\!]$, takes an existing interactive system and an agent language expression and produces a new interactive system. The agent language description represents either the synchronization or interleaving of existing agents, interleaved with an additional 3-part description of a new agent (internal, external, communication specification), or a completely new 3-part description of an agent. The following is a BNF-like description of the agent language syntax. Square brackets are used to indicate an item which is optional.

AgExp ::= **agent** *AgentID* – synchronization
synchronizes *AgentIDList*
[**with** *3PartSpec*]
endagent *AgentID*
| **agent** *AgentID* – interleaving
interleaves *AgentIDList*
[**with** *3PartSpec*]
endagent *AgentID*
| **agent** *AgentID* – 3-part specification
3PartSpec
endagent *AgentID*

The system semantic function $\mathcal{S}[\![\_]\!]$ is defined structurally over the elements in *AgExp*. For synchronized combination, the expression

agent *A1*
synchronizes *AS*
with *Spec*
endagent *A1*

maps the (fresh) agent identifier *A1* to the synchronous composition of the agents indicated by the sequence of (distinct) agent identifiers *AS*, if such a composition is allowed by $composeall_{sync}$. This may then be interleaved with the agent defined by the 3-part specification *Spec*, according to the semantic operator $\mathcal{A}g[\![\_]\!]$ discussed later.

$$
\begin{array}{|l}
\mathcal{S}[\![\ \_\ ]\!] : (IntSys \times AgEx\rho) \nrightarrow IntSys \\
\hline
\forall\, A1 : AgentID;\ AS : \mathrm{seq}_1\, AgentID;\ \rho : IntSys \\
\quad |\ (\ A1 \notin \mathrm{dom}\,\rho \\
\qquad \wedge\ (AS \fatsemi \rho) \in \mathrm{dom}\, composeall_{sync}) \\
\quad \bullet\ \mathcal{S}[\![\left(\begin{array}{l} \textbf{agent}\ A1 \\ \textbf{synchronizes}\ AS \\ \textbf{endagent}\ A1 \end{array} \right)]\!] = \\
\qquad \rho \oplus \{A1 \mapsto composeall_{sync}(AS \fatsemi \rho)\} \\
\forall\, A1 : AgentID;\ AS : \mathrm{seq}_1\, AgentID;\ Spec : 3PartSpec\ \rho : IntSys \\
\quad |\ (\ A1 \notin \mathrm{dom}\,\rho \\
\qquad \wedge\ AS \fatsemi \rho \in \mathrm{dom}\, composeall_{sync} \\
\qquad \wedge\ (composeall_{sync}(AS \fatsemi \rho), \mathcal{A}g[\![\ Spec\ ]\!]) \in \mathrm{dom}\, compose_{int}) \\
\quad \bullet\ \mathcal{S}[\![\left(\begin{array}{l} \textbf{agent}\ A1 \\ \textbf{synchronizes}\ AS \\ \textbf{with}\ Spec \\ \textbf{endagent}\ A1 \end{array} \right)]\!] = \\
\qquad \rho \oplus \{A1 \mapsto compose_{int}(composeall_{sync}(AS \fatsemi \rho), \mathcal{A}g[\![\ Spec\ ]\!])\}
\end{array}
$$

Note that because of the associativity of $compose_{sync}$ (Theorem 5.5), the order of the agent identifiers in AS does not matter.

For interleaved combination, the expression

agent $A1$
interleaves AS
endagent $A1$

maps the fresh identifier $A1$ to the interleaved product of the known agent definitions in AS and the 3-part specification $Spec$, if given.

$$\begin{array}{l}
\forall A1 : AgentID;\ AS : \mathrm{seq}_1\ AgentID;\ \rho : IntSys \\
\mid\ (\quad A1 \notin \mathrm{dom}\,\rho \\
\quad \wedge\ AS \mathbin{;} \rho \in \mathrm{dom}\ composeall_{int}) \\
\bullet\ \mathcal{S}[\![\begin{pmatrix} \textbf{agent}\ A1 \\ \textbf{interleaves}\ AS \\ \textbf{with}\ Spec \\ \textbf{endagent}\ A1 \end{pmatrix}]\!] = \rho \oplus \{A1 \mapsto composeall_{int}(AS \mathbin{;} \rho)\}
\end{array}$$

$$\begin{array}{l}
\forall A1 : AgentID;\ AS : \mathrm{seq}_1\ AgentID;\ Spec : 3PartSpec;\ \rho : IntSys \\
\mid\ (\quad A1 \notin \mathrm{dom}\,\rho \\
\quad \wedge\ (AS \mathbin{;} \rho) \frown \mathcal{A}g[\![\ Spec\ ]\!] \in \mathrm{dom}\ composeall_{int}) \\
\bullet\ \mathcal{S}[\![\begin{pmatrix} \textbf{agent}\ A1 \\ \textbf{interleaves}\ AS \\ \textbf{with}\ Spec \\ \textbf{endagent}\ A1 \end{pmatrix}]\!] = \\
\quad \rho \oplus \{A1 \mapsto composeall_{int}((AS \mathbin{;} \rho) \frown \mathcal{A}g[\![\ Spec\ ]\!])\}
\end{array}$$

Note that because of the associativity of $compose_{int}$ (Theorem 5.4), the order of the agent identifiers in AS does not matter.

A stand alone specification of an agent,

agent $A1$
Spec
endagent $A1$

maps the fresh identifier $A1$ to the agent $\mathcal{A}g[\![\ Spec\ ]\!]$.

$$\begin{array}{l}
\forall A1 : AgentID;\ Spec : 3PartSpec;\ \rho : IntSys \\
\mid\ (\quad A1 \notin \mathrm{dom}\,\rho \\
\quad \wedge\ Spec \in \mathrm{dom}\,\mathcal{A}g[\![\ _\ ]\!]) \\
\bullet\ \mathcal{S}[\![\begin{pmatrix} \textbf{agent}\ A1 \\ Spec \\ \textbf{endagent}\ A1 \end{pmatrix}]\!] = \rho \oplus \{A1 \mapsto \mathcal{A}g[\![\ Spec\ ]\!]\}
\end{array}$$

The semantic function $\mathcal{A}g[\![\ \_\ ]\!]$ maps a 3-part description from the agent language to its representative element in *Agent*. This three part description of an agent is given by separate internal, external and communication languages. The three part specification of an agent is given by an internal, external and communication language.

$$\begin{array}{rl}
3PartSpec ::= & \textbf{internal}\ IExp \\
& \textbf{communication}\ CExp \\
& \textbf{external}\ EExp
\end{array}$$

The agent semantic operator, $\mathcal{A}g[\![\ \_\ ]\!]$, is defined in terms of semantic operators for each of the sublanguages. The domain of $\mathcal{A}g[\![\ \_\ ]\!]$ is the set of combinations of internal, communication and external expressions which yield a valid agent description when they are mapped to their respective specifications in the model.

$$
\begin{array}{l}
\mathcal{A}g[\![\ \_\ ]\!] : 3PartSpec \nrightarrow Agent \\
\mathcal{I}[\![\ \_\ ]\!] : IExp \rightarrow InternalSpec \\
\mathcal{C}[\![\ \_\ ]\!] : CExp \rightarrow Communication \\
\mathcal{E}[\![\ \_\ ]\!] : EExp \rightarrow ExternalSpec \\
\hline
\mathrm{dom}\ \mathcal{A}g[\![\ \_\ ]\!] = \{\ IE : IExp;\ CE : CExp;\ EE : EExp;\ Agent \\
\qquad \mid\ (\ \theta InternalSpec = \mathcal{I}[\![\ IE\ ]\!] \\
\qquad\quad \wedge\ \theta Communication = \mathcal{C}[\![\ CE\ ]\!] \\
\qquad\quad \wedge\ \theta ExternalSpec = \mathcal{E}[\![\ EE\ ]\!]) \\
\qquad\quad \bullet \left(\begin{array}{l} \textbf{internal}\ IE \\ \textbf{communication}\ CE \\ \textbf{external}\ EE \end{array} \right) \} \\
\mathcal{A}g[\![\left(\begin{array}{l} \textbf{internal}\ IE \\ \textbf{communication}\ CE \\ \textbf{external}\ EE \end{array} \right)]\!] = \mu\, Agent \\
\qquad\quad \bullet\ (\ \theta InternalSpec = \mathcal{I}[\![\ IE\ ]\!] \\
\qquad\quad\quad \wedge\ \theta Communication = \mathcal{C}[\![\ CE\ ]\!] \\
\qquad\quad\quad \wedge\ \theta ExternalSpec = \mathcal{E}[\![\ EE\ ]\!])
\end{array}
$$

The template for this part of the agent language is shown below.

agent *AgentID*
internal
  types
    typedeclarations
  attributes
    attribute value bindings
  invariant
    predicate on state bindings
  initially
    predicate on state bindings
  operations
  op1(*typed argument list*)
    changes (*explicit framing condition*)
    pre *precondition on state and arguments*
    post *postcondition on before/after state and arguments*
  op2 ···
  ⋮
communication

inputs *typed input channels*
outputs *typed output channels*
τ *internal events*
external
constructive trace description
satisfying
trace predicate
endagent *AgentID*

For the body of this thesis, we feel there is a strong enough intuitive link between these languages and their models to warrant passing over the detail of the denotational semantics. We will, however, go into more depth on the external language.

6.2.1 A language for external specifications

There is a distinction about how an external specifications can be produced—*explicitly* via a constructive language of traces, or *implicitly* via a predicate language on traces. The constructive language of traces is usually what is provided in the notation of a process algebra, such as CSP. In this language, the constructors provide a way to build up large process specifications in terms of smaller and simpler processes. For example, the construction

$$a \rightarrow P$$

is supposed to represent the external specification which first engages in event a and then behaves like P. The complete constructive language is very similar to a subset of CSP as defined by Hoare [82].

ConEExp ::=	$\mathbf{stop}\langle\langle \mathbf{P}\ Event \rangle\rangle$	– deadlock
	$\mid \mathbf{run}\langle\langle \mathbf{P}\ Event \rangle\rangle$	– the total behaviour
	$\mid \mathbf{skip}\langle\langle \mathbf{P}\ Event \rangle\rangle$	– successful termination
	$\mid \langle\langle Event \rangle\rangle \rightarrow ConEExp$	– prefix composition
	$\mid ConEExp \,\Box\, ConEExp$	– choice composition
	$\mid ConEExp \,;\, ConEExp$	– sequential composition
	$\mid ConEExp \parallel ConEExp$	– synchronous parallel composition
	$\mid ConEExp \,\|\|\, ConEExp$	– asynchronous parallel composition
	$\mid f(ConEExp)$	– process relabelling
	$\mid \mu X : \langle\langle \mathbf{P}\ Event \rangle\rangle \bullet F(X)$	– guarded recursion

These syntactic constructs are mapped to elements in the external specification space via the semantic operator $\mathcal{E}_{con}[\![\ ]\!]$. The function $\mathcal{E}_{con}[\![\ ]\!]$ is defined in terms of two functions, one which yields the alphabet of the construct, $\alpha[\![\ ]\!]$ and one which yields the trace set, $\mathcal{T}[\![\ ]\!]$.

$$
\begin{array}{|l}
\mathcal{E}_{con}[\![\ \_\ ]\!] : ConEExp \rightarrow ExternalSpec \\
\alpha[\![\ \_\ ]\!] : ConEExp \rightarrow \mathbb{P}\ Event \\
\mathcal{T}[\![\ \_\ ]\!] : ConEExp \rightarrow \mathbb{P}\ \mathrm{seq}\ Event \\
\hline
(\mathcal{E}_{con}[\![\ ES\ ]\!]).alphabet = \alpha[\![\ ES\ ]\!] \\
(\mathcal{E}_{con}[\![\ ES\ ]\!]).traces = \mathcal{T}[\![\ ES\ ]\!]
\end{array}
$$

We define the prefix composition operator as follows.

$$
\begin{array}{|l}
\_ \rightarrow \_ : (Event \times ConEExp) \nrightarrow ConEExp \\
\hline
\mathrm{dom}(\_ \rightarrow \_) = \{\ e : Event;\ P : ConEExp \\
\qquad\qquad \mid\ e \in \alpha[\![\ P\ ]\!] \\
\qquad\qquad \bullet\ (e, P)\ \} \\
\alpha[\![\ e \rightarrow P\ ]\!] = \alpha[\![\ P\ ]\!] \\
\mathcal{T}[\![\ e \rightarrow P\ ]\!] = \langle\ \rangle \cup \{\ t : \mathcal{T}[\![\ P\ ]\!] \bullet \langle e \rangle \frown t\ \}
\end{array}
$$

As done by Hoare [82, p. 134], we introduce some syntactic conventions to handle input and outpnt. The expression

$$c?m \rightarrow P_m$$

is equivalent to the choice of every possible message that can occur along channel c followed by the behaviour of some predefined description given by P_m. For example, if there were only two messages that could occur on channel c, say m_1 and m_2, then we would have the following syntactic equivalence.

$$
\begin{aligned}
c?m \rightarrow P_m \stackrel{def}{=}\ & (c.m_1 \rightarrow P_{m_1} \\
& \Box \\
& c.m_2 \rightarrow P_{m_2})
\end{aligned}
$$

For output, we will nse $c!m \rightarrow P$ as a syntactic equivalent for $(c, m) \rightarrow P$. For both input and outpnt, the arguments for messages will not be given with type information, as it is assumed the order and types of the arguments for any given message is as defined in the internal and commnnication specifications.

In Appendix C we give the full denotational semantics for this constructive trace specification language, along with an operational semantics to aid in the intuition behind each constructor.

The advantage of a constructive language for the external specification is that it can lead naturally to an implementation. Whereas this is a definite advantage if the desire is to produce a rapid prototyping tool from the agent language, or even a full-blown programming environment, it is not the only purpose of the agent language, nor is it the primary purpose. The primary pnrpose is to be ahle to

provide a description of the components of an interactive system in such a way as to satisfy the constraints imposed by interactive properties such as those described in Chapter 4. With that purpose in mind, we admit that it is not always so simple, or possible, to describe constraints on the external behaviour using the constructive trace language.

An example of an implicit description of an external specification would be by formulation of predicates on the trace set. In CSP, these are referred to as behavioural specifications on the traces[46]. They are used as criteria for judging whether a given CSP expression satisfies some requirement. In the external specification of agents, we will allow trace predicates to describe additional constraints on the communication of events along with the constructive language.

Providing a very powerful predicate language, such as the modal mu-calculus introduced by Pratt [131] and Kozen [95], is beyond the scope of this thesis. Just as we have given a simple example of incorporating a constructive language within the agent model, so motivating further work on incorporating more complex languages, we also provide a simple example of a predicate language and leave it open to incorporate more complex languages. A trace predicate, then, is simply a set of functions with a prefix closed domain.

$$ImpEExp : \mathbf{P}(\mathbf{P}\, Event \times (\mathrm{seq}\, Event \rightarrow Bool))$$

$$(A, TP) \in ImpEExp \Leftrightarrow \mathrm{prefix\_closed}_{A}(TP(\!|\ \mathbf{true}\ |\!))$$

Just as there was a semantic function, $\mathcal{E}_{con}[\![\ \_\ ]\!]$, which mapped the constructive external language to the *ExternalSpec*, we will also have a function, $\mathcal{E}_{imp}[\!|\ \_\ |\!]$ to map the predicate language to an *ExternalSpec* element.

$$\mathcal{E}_{imp}[\!|\ \_\ |\!] : ImpEExp \rightarrow ExternalSpec$$

The overall external specification is obtained by intersecting the trace sets derived from the explicit constructive language and the implicit trace predicate language.

$$ESpecLanguage ::= ConEExp \times ImpEExp$$

$$\mathcal{E}[\![\ \_\ ]\!] : ESpecLanguage \nrightarrow ExternalSpec$$

$$\mathrm{dom}\, \mathcal{E}[\![\ =\ ]\!] \{\ A : \mathbf{P}\, Event;\ con : ConEExp;\ imp : ImpEExp \mid (\ \alpha[\![\ con\ ]\!] = A \wedge fst(imp) = A) \bullet (con, imp)\ \}$$

$$\mathcal{E}[\![\ (imp, con)\ ]\!].alphabet = \alpha[\![\ con\ ]\!]$$

$$\mathcal{E}[\![\ (imp, con)\ ]\!].traces = \mathcal{T}[\![\ con\ ]\!] \cap \mathcal{E}_{imp}[\!|\ imp\ |\!].traces$$

6.3 Using the agent language

Our presentation of the agent model will be aided by some examples. We would like to demonstrate in these examples how the definition of an agent in the new language more closely matches the way a designer understands it. The purpose of these examples is both to make clearer the meaning of the language and to show how various development techniques can be used to build up agent descriptions incrementally. The first example is adapted from Took's thesis and involves the description of a very crude nuclear reactor. The second example involves a description of parts of a simple windowing system.

6.3.1 A toy reactor

Took [163] describes a primitive nuclear reactor, in which there are damping rods to control the reaction and coolant to take away the heat generated by the reaction. There is only one relevant attribute for the rods which indicates whether they are up, enabling the reaction, or down, disabling the reaction. Initially, the rods are down. Two operations can be performed on the rods, either lifting them or dropping them, after which the rods are up or down, respectively. The rods respond to being lifted or dropped by informing some other agent of the rod position after the change. The agent definition of the rods is given below.

agent *rods*
internal
types
Rodpos ::= *up* | *down*
attributes
position : *Rodpos*
invariant
true
initially
position = *down*
operations
lift()
changes (*position*)
pre *true*
post *position'* = *up*
drop()
changes (*position*)
pre *true*
post *position'* = *down*
inform(*pos* : *Rodpos*)
pre *pos* = *position*
communication

```
    inputs rodin : lift(), drop()
    outputs rodout : inform(pos : Rodpos)
    τ :
external
    μ X • ((rodin, lift()) → rodout!inform(pos)→
           (rodin, drop()) → rodout!inform(pos) → X)
    satisfying
        true
endagent rods
```

This simple example can be used to explain the notation and several conventions we will adopt in its use. There is no state invariant (beyond that implied by the typing information of the state attributes) and we have indicated this by the predicate *true* in the invariants section of the definition. The pre- and postconditions for some operations are also listed as *true*, meaning they are as weak as possible. A *true* predicate in the satisfaction predicate for the external description means that there are no further constraints on the external definition beyond those imposed by the constructive definition and implied by the internal specification. In the future, we will usually omit these sections with predicates that are *true*.

The communication description lists the input and output channels together with the messages that can be sent along those channels. Any internal messages would have been listed after the internal channel identifier τ. There are no internal messages in this description. In the future, if there are no messages associated to a channel, it need not appear in the description. If there are no input or output events for the agent, then that section can be omitted as well.

The external specification of the rods shows the hybrid nature of the agent language. We intend to prohibit two consecntive lifts or drops of the rods. We could have easily defined the preconditions for the operations for the *lift*() and *drop*() messages to accommodate this. However, we chose in this example to use the constructive trace langnage of the external specification to express this constraint. Though it does not matter in such a simple example, there may be reasons to avoid expressing such ordering constraints by definition of preconditions. We also enforce the constraint that an output message informing of the new rod position must be performed between each input message.

The coolant for the reactor is very similar to the rods (when viewed as an agent, that is). The coolant has only one attribute of interest which indicates the level of the coolant as either in or out. Initially, the coolant is in. Operations are defined to add or remove the coolant, after which the coolant is in or out, respectively. The agent definition of the coolaut is given below.

```
agent coolant
internal
    types
```

```
        CoolantLevel ::= in | out
    attributes
        level : CoolantLevel
    initially
        level = in
    operations
    add()
        changes (level)
        pre level = out
        post level' = in
    remove()
        changes (level)
        pre level = in
        post level' = out
    inform(l : CoolantLevel)
        pre l = level
communication
    inputs rodin : add(), remove()
    outputs rodout : inform(l : CoolantLevel)
external
    satisfying
        ∀ t : traces
          • #(t | {rodout}) ≤ #(t | {rodin}) ≤ #(t | {rodout}) + 1
endagent coolant
```

Again, no two *add*() or *remove*() messages can be received consecutively. In this example we have used the precondition technique to specify this constraint. The external description need only ensure that each input event is followed by an *inform* event on the *rodout* channel, which is described by a satisfaction predicate on the trace set. The lack of a constructive external description means that its constraint is as weak as possible, equivalent to $\mathbf{run}_A$, where A is the alphabet of the agent.

The reactor agent interleaves the rod and coolant agent with the additional constraint that at no time can the level of the coolant be *out* whilst the damping rods are *up*. The interleaved combination of *rods* and *coolant* does not prohibit this, and so we must add this in as a separate invariant of the interleaved combination.

```
agent reactor
interleaves (rods, coolant)
with internal
    invariant
        ¬ (rods = up ∧ level = out)
endagent reactor
```

The behaviour of the agent *reactor* is any interleaving of the separate behaviours of the rod and coolant agents except for those which would lead to the forbidden

state.

6.3.2 Some input devices

The keyboard

The keyboard is composed of a collection of buttons, one for each labelled key. A button is a simple transducer, converting inputs of presses and releases by the user to outputs of ups and downs to the window manager. Such a button is defined below. It has one attribute of interest representing the status of the button (*up* or *down*). Two input operations correspond to the press action and the release action. After each, the new button status is sent as a response to the window manager.

agent *button*
internal
  types
    $BStatus ::= up \mid down$
  attributes
    $bstatus : BStatus$
  initially
    $bstatus = up$
  operations
  press()
    changes ($bstatus$)
    pre $bstatus = up$
    post $bstatus' = down$
  release()
    changes ($bstatus$)
    pre $bstatus = down$
    post $bstatus' = up$
  inform($bs : BStatus$)
    changes ()
    pre $bs = bstatus$
communication
  inputs $buttin$: *press*(), *release*()
  outputs $buttout$: *inform*($bs : BStatus$)
external
  $\mu X \bullet buttin?x \rightarrow buttout!inform(b) \rightarrow X$
endagent *button*

We can define a relabelling operation on agents, so that for any agent identifier $A1$, the expression

$$A1[old_1, old_2, \ldots / new_1, new_2, \ldots]$$

is mapped to the agent identified by $A1$ with the name old_1 replaced by new_1, old_2 replaced by new_2, etc. We assume that the mapping from old names to new names is injective. Furthermore, for any agent identifier $A2$ we will write $new::A2$ as shorthand for renaming all attributes and channels with the prefix *new* added. The keyboard would be represented as the synchronized combination of keys, each of which behaves as defined by *button* with the attribute and input and output channels appropriately renamed. We could have chosen to either synchronize or interleave in this example since they would both result in the same agent definition. We choose to synchronize in this case because it corresponds with our notion of the keys as separate entities on the keyboard. In the design space notation of Card, Mackinlay and Robertson [31], this composition is classified as "layout composition", referring to the collocation of two or more devices (the keys in our example) on different places of a common panel (the keyboard in our example).

agent *keyboard*
synchronizes $\langle$ *a::button*, *b::button*, ...$\rangle$
endagent *keyboard*

The mouse

The mouse also makes use of the *button* agent for each of its buttons. In addition, the mouse can transform movements in a 2-dimensional plane to values in the *Pixel* plane. This transformation is an unchangeable attribute of the mouse agent, *res*, and it corresponds to what Card, Mackinlay and Robertson refer to as a "resolution function". Below, we describe the movement portion of a mouse as the agent *MouseMove*.

agent *MouseMove*
internal
  types
    $mousexmax, mouseymax : \mathbb{N}$
    $MousePlane == 0..mousexmax \times 0..mouseymax$
  attributes
    $currentmove : Pixel$
    $res : MousePlane \rightarrow Pixel$
  operations
  sweep(*mdelta* : *MousePlane*)
    changes (*currentmove*)
    post $currentmove' = res(mdelta)$
  mousemove(*pdelta* : *Pixel*)
    changes ()
    pre $pdelta = currentmove$
communication
  inputs *gesture* : *sweep*(*mdelta* : *MousePlane*)
  outputs *mouseout* : *mousemove*(*pdelta* : *Pixel*)

external
$\mu X \bullet gesture?x \rightarrow mouseout!mousemove(pd) \rightarrow X$
endagent *MouseMove*

The mouse agent is the synchronized combination of the movement agent and the buttons agents, of which we assume there are three.

agent *Mouse*
synchronizes $\langle 1_m::button, 2_m::button, 3_m::button, MouseMove \rangle$
endagent *Mouse*

Card, Mackinlay and Robertson have prodnced a generative design space for describing a large class of inpnt devices. The example of the keyboard and monse above show how we can realize some points in that space and it wonld be an interesting exercise to generate more of that space in the agent langnage. Our limited experience suggests that the slightly unnatnral way of expressing stimnlns-response behaviour may be a limiting factor in the snccess of such an exercise.

6.3.3 A window

We can represent an individnal window in a windowing system based on the description given in Section 6.1. First we will allow some global type definitions which will be available to all agent definitions in this section. These will allow us to talk ahout points in the window space in terms of pixels, and images in terms of pixel maps.

$xmax, ymax : \mathbb{N}$
$Pixel == 0..xmax \times 0..ymax$
$Bit ::= 0 \mid 1$
$PixelMap == PIXEL \pfun Bit$

We will assume appropriately defined operations on elements in *Pixel*, such as addition and a natural ordering $\leq$. We give an incremental description of the window which mirrors the development method used with Z. The window state is first, indicating the status of the window as open or closed, the icon associated to the window when closed and its position, and the size of the window when open and its position. Initially, the window is open, and the other attribntes are set to some default value.

agent *WindowState*
internal
types
$winsize_{def} : Pixel$
$icon_{def}, blank : PixelMap$
attributes

```
        icon, contents : PixelMap
        iconpos, winpos, winsize : Pixel
        status : open | closed
    invariant
        winpos + winsize ≤ (xmax, ymax)
    initially
        status = open
        winpos = iconpos = (0, 0)
        winsize = winsize_def
        icon = icon_def
        contents = blank
    operations
    showwin(c : PixelMap)
        changes ()
        pre  status = open
             c = contents
    showicon(i : PixelMap)
        changes ()
        pre  status = closed
             i = icon
communication
    outputs
        winout : showwin(c : PixelMap), showicon(i : PixelMap)
endagent WindowState
```

The agent *WindowState* will be able to continuously send output messages of its blank window contents. Because the messages *showwin*(c) and *showicon*(i) will be used by many other agents to be defined below which interleave with *WindowState*, we must show in each case that the effect on attribntes not in common is the identity transition. In practice, the messages that are shared between agents represent output responses and these are usually defined as identity transitions, so we will satisfy the constraint trivially.

Opening and closing the window is described by interleaving *WindowState* with an agent description of those operations.

```
agent OpenClose
interleaves ⟨WindowState⟩
with
internal
    operations
    open()
        changes (status)
        pre status = closed
        post status' = open
    close()
```

```
        changes (status)
        pre status = open
        post status' = closed
communication
    inputs
        winin : open(), close()
endagent OpenClose
```

Moving the window is described as a single operation. The postcondition covers the two cases in which the window is open or closed.

```
agent Move
interleaves ⟨WindowState⟩
with
internal
    operations
    move(pos : Pixel)
        changes (winpos, iconpos)
        post  status = closed ⇒  ( winpos' = pos
                                 ∧ iconpos' = iconpos)
              status = open ⇒     ( iconpos' = pos
                                 ∧ winpos' = winnpos)
communication
    inputs
        winin : move(pos : Pixel)
endagent Move
```

Finally, resizing a window can only occur when the window is open.

```
agent Resize
interleaves ⟨WindowState⟩
with
internal
    operations
    resize(s : Pixel)
        changes (winsize)
        pre status = open
        post winsize' = s
communication
    inputs winin : resize(s : Pixel)
endagent Resize
```

All of the above agents can be combined to give an agent description of a window. We add the constraint overall that requires each event on the *winin* channel to be followed by an event on the *winout* channel. The event on the output channel either indicates the iconic form of the window or displays the contents, depending

on whether the window is closed or open. In order to guarantee this behaviour we introduce internal events $(\tau, isopen())$ and $(\tau, isclosed())$ to decide which view of the window is output as a response to the last input.

agent *Window*
interleaves $\{OpenClose, Move, Resize\}$
with
internal
  operations
  $isopen()$
    pre $status = open$
  $isclosed()$
    pre $status = closed$
communication
  τ : $isopen(), isclosed()$
external
  $winout!showin(c) \rightarrow$
  $\mu X \bullet$ $(winin?x \rightarrow$
    $((\tau, isopen()) \rightarrow winout!showwin(c) \rightarrow X$
    $\Box$
    $(\tau, isclosed()) \rightarrow winout!showicon(c) \rightarrow X))$
endagent *Window*

A window manager

Having defined the individual window, we want to investigate how to define the window manager as an agent which synchronizes with and coordinates the activity of the individual windows. Figure 6.3 gives a graphical view of the agent relationship we want to capture.

Interaction with any window is coordinated via the window manager. In the manager we describe here, we will assume that windows are nonoverlapping in *Pixel* space and that commands to the window manager are directed to the currently selected window (if there is one). The currently selected window is determined by the position of the mouse cursor. These assumptions have been made to make the description simpler. We could easily relax these restrictions, but the corresponding description of the window manager would be more complex. We will not give a complete description of the window manager either. Instead, we will give an example of how it coordinates activity by describing how the window opening command is defined as an operation on the window manager.

The attributes of concern for the window manager include:

- the position of the mouse cursor;
- the set of *PixelMaps* associated to active windows in the system;

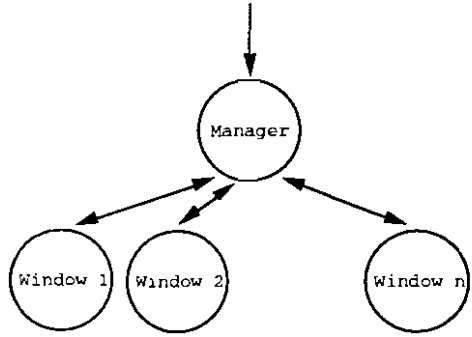

Figure 6.3: Window manager/window relationship

- a function for determining to which window a point in Pixel space belongs; and
- the currently selected window, which is at most one window.

Identification of windows will be by a set of window identifiers, as was proposed by the promotion solntion discussed earlier.

$[WinID]$

We will not concern ourselves with initialization considerations in this description, as it is not important to the example of opening windows. Below is the agent description of the attributes and invariants for the window manager. The constraint of nonoverlapping windows is embodied in the definition of *pick* as a function.

agent *WindowManager*
internal
attributes
$mousepos : Pixel$
$windows : WinID \pfun PixelMap$
$known, selected : \mathbb{P}\ WinID$
$pick : Pixel \pfun WinID$
invariant
$known = \mathrm{dom}\ windows$
$\#selected \leq 1$
$selected \subseteq known$
$pick = \{\ w : known;\ p : Pixel$
$\mid\ p \in \mathrm{dom}(windows(w))$
$\bullet\ p \mapsto w\ \}$
$selected = pick(\!|\{mousepos\}|\!)$

The window manager will receive as input messages from a potential user which can change the mouse position or open the current window. From individual windows, the manager receives as input the *showwin*(c) message that was mentioned earlier. The response message of importance in this example is the message sent to the selected window to open it. We also describe an internal message *isselected*(w : *WindID*) which decides if the window labelled by the identifier w is the currently selected window. This internal event will allow us to give the desired external description of the manager's behaviour. The following is the definition of the operations for the window manager.

operations
movemouse(dp : *Pixel*)
  changes (*mousepos*)
  post $mousepos' = mousepos + dp$
showwin(c : *PixelMap*)
  changes (*windows*, *selected*, *pick*)
  pre $selected \neq \emptyset$
  post $windows' = windows \oplus \{ s : selected \bullet s \mapsto c \}$
openwindow()
  changes ()
  pre $selected \neq \emptyset$
open()
  changes ()
isselected(w : *WinID*)
  changes ()
  pre $w \in selected$

The communication specification for the window manager has one channel *wmin* for input received from the user and one input channel from every window. Our description of the window manager is statically determined, that is, all communications channels that will ever be used must be declared at once since there is no means of adding channels.[2] We therefore define an input chanuel from every window, even if that channel is never used. Likewise, we define an ontput channel from the window manager to each window. The commnnication description for the window manager follows.

communication
  inputs
    wmin : *movemouse*(dp : *Pixel*), *openwindow*()
    $\forall w$: *WinID* $\bullet$ *w::winout* : *showwin*(c : *PixelMap*)
  outputs
    $\forall w$: *WinID* $\bullet$ *w::winin* : *open*()
  τ : *isseleeted*(w : *WinID*)

[2]It should be possible to use the agent model dynamically, allowing channels to be communicated as well, but we have not investigated this as yet

The external specification of the window manager is more complicated than we have seen previously. This is because the operations defined do not imply much ordering information. When an *openwindow*() message is sent along the *wmin* channel, it is meant to open the cnrrently selected window, determined by the mouse position. The window manager determines which window is selected by the internal event $(\tau, isselected(w : WinID))$ which binds the value of w so that the manager can send the *open*() message along the channel $w::winin$, and then receive the new pixel image for the newly opened window along the $w::winout$ chanuel, as desired. The selected window is changed by movement of the mouse. The external specification for the the window manager is given below.

$$
\begin{array}{ll}
\textbf{external} & \mu X \bullet ((\Box_{w:WinID}((\tau, isselected(w)) \rightarrow OPEN_w)) \\
 & \qquad \Box \\
 & \qquad (wmin, movemouse(dp)) \rightarrow X) \\
\textbf{where} & \\
OPEN_w = & ((wmin, openwindow()) \rightarrow (w::winin, open()) \rightarrow \\
 & \qquad (w::winout, showwin(c)) \rightarrow X) \\
\textbf{endagent} & WindowManager
\end{array}
$$

6.4 Conclusions

In this chapter we have presented a language for the description of agents which is more suited to design than the model notation of Chapter 5. The language we have chosen borrows heavily from standard Z and CSP. This has two advantages. First, the semantics of the notation is similar to the semantics for those two uotations. Second, familiarity with those notations should increase familiarity with the agent notation.

There are disadvantages, however, to our approach. The agent model is based on a stimulus-response model, but there is no natural means for expressiug the connection between stimulus and response, which should occur at the operation description level. Rather, we have had to describe all inputs (stimuli) and outputs (responses) as independent messages and leave it to the pre- and postconditions of their operation definitions or the external specification to describe auy relationship between them. Since output messages usually have no effect on the internal state of an agent (at least not in the examples we have given), the relationship between stimulus and response is left entirely up to the external specification. A more serious criticism is that this is not necessarily a flaw in the notation we have chosen but an undesired bias in the agent model itself.

This lack of stimulus-response connectedness is probably the most important featnre of the Z language that we have lost in the agent language. Z provides a very natural way of grouping stimulus and response behaviour with a state transition. Unfortunately, Z does not provide the general communication mechanism we would

like, which is why we had to abandon it. A possibility we would like to investigate would be the enhancement of Object-Z by provision of a more flexible facility for expressing object to object communication.

Despite these criticisms, the agent language is useful for descriptions of realistic interactive systems, and its hybrid approach and the composition operators do allow for a systematic description of complex behaviour, as desired.

Chapter 7

Properties of interactive systems: Part II

In this chapter we will use the refined agent model to formally express interactive properties at a more concrete level than was done in Chapter 4. In that chapter, we showed how a simple stimulus-response model could be used to formulate properties relevant to interaction with an agent. The class of properties which we could express were very abstract, much like the properties expressible in the PIE model. Properties such as predictability, consistency, honesty and trustworthiness, as expressed in the simple agent model, are not very closely linked with the interactive properties suggested by the interaction framework. The framework suggests that the *Interface* (*Input* composed with *Output*), should effectively mediate between the tasks of the *User* and the functions of the *System*.

Therefore, the task is seen as providing scope for properties which attempt to describe how the *Interface* can serve as a good mediator. We define the role of task analysis in the reformulation of interactive properties. We define the output of task analysis as an identification of *System* and *Interface* attributes which are understood by the *User* to portray the effect of a given task. The *System* attributes form a *result template* of information relevant to the achieved goals of the interaction. The *Interface* attributes form a *display template* of visible information that the *User* understands in terms of how it reflects the values of attributes in the result template. Templates are used to restrict access to information of an agent by any of its observers. The interactive properties we will formulate in this chapter will be relationships between the input histories and a result or display template value history.

Since the properties are formulated with respect to the result and display templates, they are task-specific. Hence, we will be able to describe, for example, what it means for a graphics package to be display predictable with respect to selection of objects or show how a direct manipulation operating system interface lacks hon-

esty in supporting the task of directory creation. Though we will not present any of the examples in this chapter in complete detail, we will present enough of the detail within the agent language so that an informal understanding of the properties can be gained with confidence that a completely formal description could also be provided.

The reformulation of task-specific interactive properties and the examples justify our claim that the refined agent model and language take us one step closer towards realizing a design practice guided by principled reasoning. In a design situation, the task analysis provides the result template information for each task that an agent should support. The *System* agent is then specified and can be verified to check that it satisfies certain result only properties, such as result predictability. The duty of the designer is then to determine for each task which display templates can be chosen and specified to satisfy the constraints imposed by the result-display properties, such as honesty. In analysing an existing design, the task analysis provides the display template information as well as the result template information and the design can be tested to see if it satisfies any of the interactive properties, and if not why it fails.

The main reason we will be able to speak more concretely is that the we know more about how the agent functions. The internal structure of the agent is no longer just a set of states about which we know nothing further. We can now look inside that state and describe it in terms of its attributes, and these attributes can be determined based on the level of the description necessary to address a given task. Just as the agent view of an interactive system allows us to separate its description into more manageable computational units, so too does the attribute and template view of an agent allow us to separate its description into more manageable perceivable units.

Overview of chapter

We begin in Section 7.1 with a discussion of the relationsbip between display and result and how previous formalisms have treated this relationship. In Section 7.2 we define a template as a means of isolating aspects of the display and result and relating them between composed agents. The remaiuing sections of this chapter reformulate the properties discussed in Chapter 4 using the display and result template information to derive more task-relevant definitions. Several examples are discussed along the way to make clear the use of the reformulated interactive properties, especially in the analysis of existing designs.

7.1 Relating Display and Result

As we have said, tbe purpose of interaction between human and computer is for the user to attain certain goals within some application domain. Tbe *results* of interaction, then, are the achieved goals. The product of a task analysis is a description of the user's assumed goals, or an identificatiou of the desired results of interaction. A principled, nser-centred design methodology begins with this infomation and describes the system initially as an abstract machine whose state description is well suited to the task structure of the user. Hence, there sbould be a close match between the results as defined by the task analysis and the state of the system.

Even for systems which have not been designed initially in this way, it is possible to reverse engineer an abstract description of the system which concentrates on task analytic information. This is the intent behind the examples of this chapter and the next, in which properties are exemplified by examining how they are missing in existing interactive systems. The motivation for this kind of analysis comes from the scenario methodology [174, 175], which provides examples of user behaviour with real systems as a means of cross-fertilization of different modelling domains in HCI research.

It is outside the scope of this thesis to consider any further the description of the *User*. Since we assume it is possible to produce a description of the *System* in line with the results desired by the *User*, we will use the term result from now on to refer to the end products of user interaction in terms of the state of the *System*. The agent model and its language deal directly with building up a description of the *System* state.

The user does not see the state of the system directly; rather, parts of it are rendered in the display space. A *display* is the immediate and perceivable featnre of the system from which the user must interpret the relevant features of the system state. The user constructs a relationship between the perceivable information of the display and the desired but hidden information on the result of the system. Having established this crucial link between the display and the result, we can investigate means for formalizing the relationship.

As mentioned in Chapter 3, the red-PIE model is an extension to the simple PIE model which allows for an abstract discussion of the relationship between result and display. In the red-PIE model, we introduce separate functions on the effects space which separately extract the display and result information, as shown again in Figure 7.1.

In the same vein, the Sufrin and He model of processes was enhanced to discuss the relationship between results and views. In that model, an interactive process is built on top of the original process (represented by the schema type $SandHProcess[S]$ in Chapter 6) by including mappings from tbe state space to the display and result spaces and events whicb trigger the calculation of the display

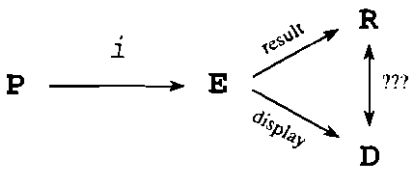

Figure 7.1: The red-PIE model.

and result from the value of the state at the time of the event.

In the agent model, we adopt a more constructive approach in order to relate better with actual design. Consequently, we must be more explicit about the display and the result than is necessary in either of the other two more abstract models. Looking to the interaction framework, we see that the results as we have described above are derived from the *System* and the displays from the *Interface*, or more specifically, the *Output*. Relationships between display and result, then, are relationships between agents. We derived one such relationship in Section 4.3, called correspondence. The key to correspondence earlier was the definition of a relationship between the state spaces of two agents, called the *retrieve* relation. The simple agent model provided no way to derive the *retrieve* relation. Attributes and templates will provide such a mechanism that can be derived from empirical evidence. The attributes used in the description of an agent are then viewed as design decisions by the specifier and the justification for those decisions comes from the templates obtained from task analysis. We delay a fuller treatment of correspondence until the next chapter.

Properties like predictability, consistency and honesty were also expressible in the simple agent model. These properties relied on a distinction between internal state behaviour and external response behaviour and classified systems in terms of the equivalence and indistinguishability of stimuli with respect to the state changes or responses they determined. Using a similar approach in this chapter, we will be able to define these properties and others in terms of display and result templates on agents composed as described by the interaction framework. The result templates will highlight relevant parts of the *System* state, whereas display templates will highlight information to be presented potentially as channels of communication to the user. The advantage of this approach will be that it will be made explicit what assumptions an agent design makes about the user's understanding of the interaction, and these assumptions can then be tested to see if they lead to more usable systems.

In sum, the properties we are interested in expressing in this chapter deal with the relationship between the display and the result, as indicated by the additional

unnamed arrow in Figure 7.1. They will be given as properties dictated by the design of an agent-based system and they are intended as gauges for usability in situations where the nser depends on the display to determine the underlying result of the interaction or when the result knowledge of the user is needed to determine the meaning of the display.

7.2 Templates

The reason for introducing attributes is to enahle restricted views of an agent. The restricted views are called *templates* and were first discussed by Harrison, Roast and Wright [74] as a means of focusing attention on parts of the result or display in order to formalize realistic relationships between them. For example, when using a text editor within a single window on a multi-windowing workstation, the user does not usually care about the contents of other windows. They may not even pay attention to some of the information in the text editor window. If a user relies upon certain properties of the interface, such as honesty, then it is important to be able to narrow the scope of information in the system over which the honesty property holds. The importance of templates in design and analysis is that they are not generated by our formalism; rather, they are regarded as input to the formalism which leads to a truly formal *method* of design and analysis.[1]

We mentioned in Chapter 4 how we can view the properties such as predictability in terms of the demands they make on the user. For example, in a graphic drawing package, the user must remember the order in which objects were drawn for predicting the consequence of clicking to select in a region of the 'canvas' occupied by overlapping objects. A Macintosh user must remember the value of the 'items' indicator in the top left-hand corner of a folder window to determine whether clicking ou 'New Folder' produced a new folder, siuce the new icon may not appear in the window [137]. In text editing, I don't usually pay attention to the position of the mouse pointer when I am typing, so the mouse pointer is not in my display template when I predict the result of inserting a character. This does canse prohlems—the infamous 'unselected windows' scenario—when the mouse is accidentally nudged and my keyboard input is suddenly directed to the wrong window. Notice how the description of each of these examples is given within the scope of some task the user wishes to perform.

[1]Most of what is commonly referred to as *formal methods* does not actually represent any method, but just a notation with a formal semantics

7.2.1 Agent restriction

A template defines a restricted view of an agent by limiting the information known about the underlying state. Since all that is known about individual points in the state space is contained in the attribute-value mappings, we can define a template as a nonempty, finite set of attributes.

$$Template == \mathbb{F}_1 \mathcal{A}$$

A template defines that part of the internal specification which is of interest; it therefore induces a new internal specification for an agent. The attributes of the template must all be valid attributes of the original internal specification. The types of the template attributes remain the same. We only keep state information on the attributes in the template, so states of the original internal specification which agree on all attributes of the template are equated in the new internal specification. For each message of the original internal specification, and for each state transition in the operation associated to that message, restricting the view of those states to the attributes of the template yields a state transition for that message in the new internal specification. A characterizatiou of template restriction on an internal specification is given by the schema *InternalRestrict* below, in which the original internal specification is decorated with ^{I} and the induced internal specification is decorated with ^{J}.

InternalRestrict

$InternalSpec^I$

$InternalSpec^J$

$t : Template$

$t \subseteq attribs^I$

$attribs^J = t$

$type^J = t \lhd type^I$

$states^J = \{ s : states^I \bullet t \lhd s \}$

$messages^J = messages^I$

$\forall m : messages^J$

$\bullet \quad operations^J(m) = \{ (s, s') : operations^I(m) \bullet (t \lhd s) \mapsto (t \lhd s') \}$

A restriction of an agent with respect to a template of attributes is characterized by the schema *Restrict*, in which the original agent is decorated with ^{I} and the template-restricted agent is decorated with ^{J}. The only change to the agent is to its internal specification, as given by *InternalRestrict*. The communication and external specifications remain the same.

$$
\begin{array}{|l}
\textit{Restrict} \\
\hline
\textit{Agent}^I \\
\textit{Agent}^J \\
\textit{InternalRestrict} \\
\hline
\theta \textit{Communication}^I = \theta \textit{Communication}^J \\
\theta \textit{ExternalSpec}^I = \theta \textit{ExternalSpec}^J \\
\hline
\end{array}
$$

We write $A \Downarrow t$ for the agent induced by restrictiou of A to the template t.

$$
\begin{array}{|l}
\_ \Downarrow \_ : (\textit{Agent} \times \textit{Template}) \nrightarrow \textit{Agent} \\
\hline
\_ \Downarrow \_ = \{\ \textit{Restrict} \bullet (\theta \textit{Agent}^I, t) \mapsto \theta \textit{Agent}^J\ \}
\end{array}
$$

7.2.2 Result and display templates

We can restrict any agent to a template of a subset of its attributes. However, within the interaction framework, different templates take on a different purpose, and so we distinguish them by name. Templates applied to the *System* agent are called *result templates*. Templates applied to the *Interface* agent we call *display templates*.

Attributes in the description of the *User* are similar to *semantic features* as used in TAG [128] and in the knowledge analysis work of Youug and Whittington [173]. A task description for the user highlights the psychological attributes of importance. These psychological attributes are roughly equivalent to *System* attributes. Therefore, given a task, we can isolate the *System* attributes of interest for that task. So the output of some task or knowledge analysis, would yield a mapping from a set of identified tasks to result templates. A designer must then choose attributes of the *Interface*, the display template, which will relate to the result template for the task. We model the output of task or knowledge analysis by functions from some set of tasks to the result aud display templates the tasks require.

$$[\textit{TaskID}]$$

$$
\begin{array}{|l}
\textit{Rtemplate} : \textit{TaskID} \nrightarrow \textit{Template} \\
\textit{Dtemplate} : \textit{TaskID} \nrightarrow \textit{Template} \\
\hline
\mathrm{dom}\ \textit{Rtemplate} = \mathrm{dom}\ \textit{Dtemplate} \\
\textit{Rtemplate}(t) \subseteq \textit{System.attribs} \\
\textit{Dtemplate}(t) \subseteq \textit{Interface.attribs}
\end{array}
$$

From a methodological point of view, the result template information is always provided by the task analysis. Display template information is provided for analysis of an existing system, but it is not provided for the design of a new system, since that is the responsibility of the designer. However, since real design is an iterative procedure and the systems users have in the past experienced influences their understanding of the tasks they will perform with new systems, we will stick with the assumption above that task analysis provides both the display and result template information.

We can now define properties of the interactive system in terms of these task-dependent result and display templates. This provides a principled means of iterative design, because if the result and display templates defined do not satisfy the requirements of the properties we will discuss then they will have to be altered. Typically, the display templates will be altered, as they represent attempts by the designer to effectively portray the result templates at the interface. However, as the structure of tasks is seen by many HCI researchers as dynamic [157, 101, 35], we can allow for the result template information to change as well.

7.2.3 Equivalence and indistinguishability revisited

In the latter half of Chapter 4, we introduced the notions of equivalence and indistinguishability as a way of relating different program inputs for an agent. Then, we differentiated between internal and external equivalence and indistinguishability to show how certain classes of properties could be expressed as relationships between internal behaviour and external behaviour. From a software engineering point of view, the distinction between internal and external behaviour, as personified by the interpretation relations I_A^{int} and I_A^{ext}, is satisfactory because it can be used as the basis for a refinement calculus on agents, similar to the way that Whysall and McDermid use export and body specifications of objects in their object-oriented use of Z [167, 168]. From an HCI point of view, this distinction is less useful because it does not directly address the relevance of the external or internal information to the goal-directed interaction.

The result/display distinction is directly intended to address this earlier deficiency. Rather than suggest that the following formalizations of interactive properties replace the ones developed in Chapter 4, we suggest that they are complementary, moving more toward analysis of HCI considerations. We point out that the earlier properties need further investigation to make clear their contribution to refinement within the agent model, but it was not the intent of this thesis to pursue that point. Sufrin and He [158] showed how to define refinement on processes and proved the soundness of downward simulation along with a method for stepwise refinement. A similar procedure can he carried out on the agents.

Given a task t : *TaskID* and an interactive system *IS* defined as a collection

of composed agents, we want to formalize the relationship between the results and displays of *IS* with respect to the task t. If we restrict *IS* to the attributes in $Rtemplate(t)$, we will capture the result behaviour of the computer. If we restrict *IS* to the attributes in $Dtemplate(t)$, we get the display behaviour. Histories of either restricted agent are equivalent if they lead to the same possible results or displays and they are indistinguishable if further identical extensions to the histories does not betray the result or display equivalence. Below we give the formal definitions of result and display equivalence and indistinguishability.

$$
\begin{array}{|l}
\_ \approx \_ \_ : (Agent \times Template) \nrightarrow (\mathrm{seq}\, Event \leftrightarrow \mathrm{seq}\, Event) \\
\_ \equiv \_ \_ : (Agent \times Template) \nrightarrow (\mathrm{seq}\, Event \leftrightarrow \mathrm{seq}\, Event) \\
\hline
\mathrm{dom}(\_ \approx \_ \_) = \{\, A : Agent, t : \mathbf{P}\, \mathcal{A} \mid t \subseteq A.attribs \,\} \\
\mathrm{dom}(\_ \equiv \_ \_) = \mathrm{dom}(\_ \approx \_ \_) \\
p \approx_{(A,temp)} q \Leftrightarrow I^{ini}_{A \Downarrow temp}(|\{p\}|) = I^{ini}_{A \Downarrow temp}(|\{q\}|) \\
p \equiv_{(A,temp)} q \Leftrightarrow \forall r : \mathrm{seq}\, Event \bullet (p \frown r) \approx_{(A,temp)} (q \frown r)
\end{array}
$$

In situations where the agent is clear from context, we will usually abbreviate $p \approx_{(A,temp)} q$ to $p \approx_{temp} q$.

We can now use these more task-oriented definitions of equivalence and indistinguishability to discuss interactive properties similar to Chapter 4. The introduction of task information and the result and display templates allows us to define a scope to the properties discussed in Chapter 4 that was not then possible. In addition, the meanings associated to the properties will bear more significance toward the interaction between *User* and *System*, since they will be couched in terms of task and will be focussed on features relevant to the task.

In addition to defining a scope for properties, attributes allow the definition of some proof obligations. The properties we will discuss below are expressed over histories of interaction, which includes all events in which an agent participates. When an agent is restricted to a given template, some of the events in which the agent participates have no effect on the restricted state; their effect is confined to attributes outside the template and, therefore, they are independent of the template. Such independent events can be neglected in proofs over histories for a given template.

$$
\begin{array}{|l}
independent : (Agent \times Template) \nrightarrow \mathbf{P}\, Event \\
\hline
\mathrm{dom}\, independent = \{\, A : Agent, t : Template \mid t \subseteq A.attribs \,\} \\
e \in independent(A.t) \Leftrightarrow \\
\quad \#((A \Downarrow not\_t).operations(\mathrm{mess}(e))) = \#(A.operations(\mathrm{mess}(e))) \\
\qquad \textbf{where}\ not\_t = A.attribs - t
\end{array}
$$

Sometimes, the user believes some events are independent of some attributes, when in fact they are not. For example, in the 'unselected windows' scenario mentioned earlier, the problem arises because keyboard events are not independent of mouse position, though the user tends to forget this.

7.3 Predictability and Consistency

We fix on a particular instance of the task knowledge for an interactive system given by *Rtemplate* and *Dtemplate* defined earlier. A task is *result predictable* if result equivalent histories with respect to that task and its associated result template are also result indistinguishable.

$$\begin{array}{|l} ResultPredictable : \mathbb{P}(TaskID) \\ \hline task \in ResultPredictable \Leftrightarrow (\ task \in \mathrm{dom}\ Rtemplate \\ \qquad\qquad\qquad\qquad \wedge\ (\_ \approx_{rt} \_) \subseteq (\_ \equiv_{rt} \_)) \\ \quad \textbf{where}\ rt = Rtemplate(task) \end{array}$$

Technically, the above equivalence and indistinguishability should be indexed by the particular agent, say *IS* which represents the composition of a *System* and *Interface* along with the translations between them. We take the liberty of omitting explicit mention of *IS* throughout this chapter.

It is sensible that every task of interest be result predictable. Some possible reasons for a task which is not result predictable would be the task's dependence on pure randomness (rare), or an incomplete task analysis (undesirable).

A task is *display predictable* if display equivalent histories are also display indistinguishable.

$$\begin{array}{|l} DisplayPredictable : \mathbb{P}(TaskID) \\ \hline task \in DisplayPredictable \Leftrightarrow (\ task \in \mathrm{dom}\ Dtemplate \\ \qquad\qquad\qquad\qquad \wedge\ (\_ \approx_{dt} \_) \subseteq (\_ \equiv_{dt} \_)) \\ \quad \textbf{where}\ dt = Dtemplate(task) \end{array}$$

Display predictability is related to the original predictability properties formulated in the PIE except that the addition of task information has confined its scope. We can give a simple and semi-formal example of the violation of display predictability, based on the SuperPaint graphic art package for the Macintosh [143]. SuperPaint supports *layers* for the construction and manipulation of pictures. These layers are independent canvasses upon which pictures can be constructed. The two layers in SuperPaint are the paint layer, in which freehand pictures are constructed and manipulated at the lowest level of screen detail (the pixel), and the object layer, in which text, boxes, circles, etc. can be created and edited as whole entities.

The particular scenario of interest for display predictability involves the task of selection. Figure 7.2 depicts the situation presented to the user. A mouse click on

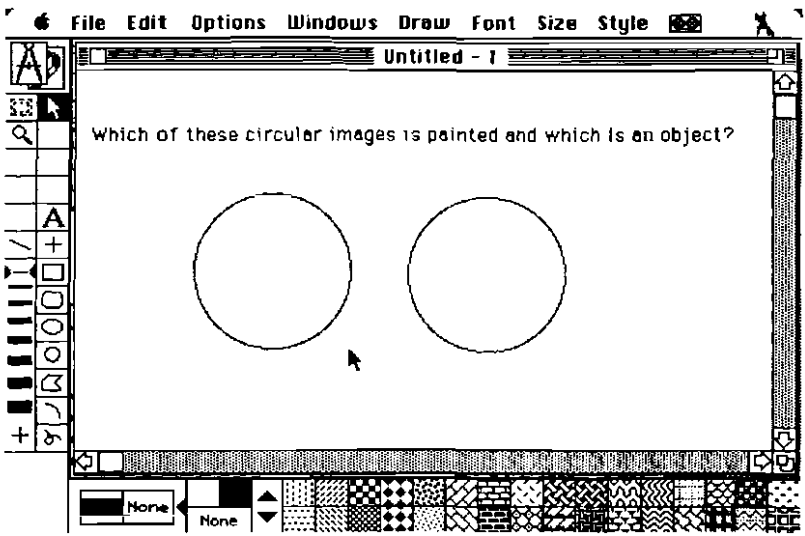

Figure 7.2: Ambiguous object selection in SuperPaint

an object will select it, so if the user knew which visual images were objects, then it would be possible to predict from a display template consisting of the mouse pointer position and the positions of the objects the effect of subsequent mouse clicks. SuperPaint does not coutain in its description of the *Interface* attributes which distinguish objects from painted pictures, so the only attribute information that is perceived via a display template by the user is the set of all visual images. Since similar looking visual images can be created in either layer, selection is not display predictable.

We can base the task analysis information for selection on the description given iu the user's manual [143]. We will first treat the different layers separately, and then examine their combination. For the paint layer, we have an agent description

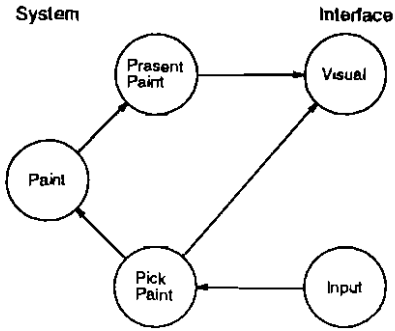

Figure 7.3: Agent diagram of the paint layer

presented graphically by Figure 7.3. The system description is contained in the agent *Paint* and it contains the result information for selection. There is only one image of interest and that is the whole painted image on some finite coordinate plane, called the canvas. Selection is done by dragging out some connected region in the canvas plane, which then selects that region for future operations. The definition of the agent *Paint* for the selection task only is given below.

agent *Paint*
  types
    $canvasxmax, canvasymax : \mathbb{N}$
    $Canvas == 0..canvasxmax \times 0..canvasymax$
    $Bit ::= black \mid white$
    $CanvasMap == Canvas \rightarrow Bit$
  attributes
    $painting : CanvasMap$
    $pselected : \mathbf{P}\ Canvas$
  invariant
    $pselected \subseteq \mathrm{dom}\ painting$
  operations
  $selectregion(region : \mathbf{P}\ Canvas)$
    changes $(pselected)$
    post
      $pselected' = region$
  $boundingbox(region : \mathbf{P}\ Canvas)$
    changes ()
    pre
      $region = pselected$
communication

inputs *paintin* : *selectregion*(*region* : $\mathbb{P}$ *Canvas*)
outputs *paintout* : *boundingbox*(*region* : $\mathbb{P}$ *Canvas*)
external
$\mu X \bullet ((paintin, selectregion(x)) \rightarrow paintout!boundingbox(y) \rightarrow X)$
endagent *Paint*

For the object layer, we have a very similar agent description as that for the paint layer, and this is given graphically in Figure 7.4. The system description is

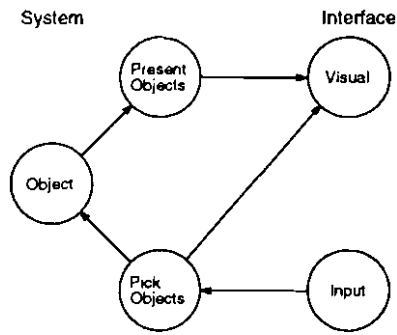

Figure 7.4: Agent diagram of the object layer

contained in the agent *Object*, and it is more complicated than the description of the agent *Paint*. It is necessary now to identify individual graphical objects and the structure implied by grouping of those objects into more complicated objects (we will ignore the further complication of levels associated to overlapping objects). We represent this structure by a straightforward hierarchy in which any object can be linked to at most one parent object. The selectable objects are only those which are not linked to a parent object. The currently selected objects are some subset of the selectable objects. We define only one selection operation, which given some subset of the selectable objects makes that set the currently selected objects. The description of *Object* corresponding to this result template of attributes is given below.

agent *Object*
types
[*ObjID*]
attributes
drawn : $\mathbb{P}$ *ObjID*
grouping : *ObjID* $\nrightarrow$ *ObjID*
objselectable : $\mathbb{P}$ *ObjID*
objselected : $\mathbb{P}$ *ObjID*

invariant
$drawn = \text{dom } grouping \cup \text{ran } grouping$
$objselectable = \text{ran } grouping - \text{dom } grouping$
$objselected \subseteq objselectable$
operations
$selectobjs(ids : \mathbf{P}\ ObjID)$
changes $(objselected)$
post
$objselected' = region$
$showselected(ids : \mathbf{P}\ ObjID)$
changes ()
pre
$ids = objselected$
communication
inputs $objectin : selectobjs(ids : \mathbf{P}\ ObjID)$
outputs $paintout : showselected(ids : \mathbf{P}\ ObjID)$
external
$\mu X \bullet ((objectin, selectobjs(x)) \rightarrow paintout!showselected(y) \rightarrow X)$
endagent *Object*

The *Visual* agent contains a mapping from some screen coordinate space, *Pixel*, to bit values, which we will restrict to simply the value *black* or *white*. The pixel map can be further divided into regions which the user may identify as an iudependent image, so that the whole pixel map is considered as a collection of smaller, possibly overlapping, pixel maps. There is an indication of the current layer (object or paint) and the current tool in that layer, both of which give the mode in which inpnt is interpreted. The mouse cursor is also located somewhere in the screen coordinate space. So, for the task of selection, we highlight information described above as a set of attributes describing the state space of the *Visual* agent.

agent *Visual*
internal
types
$screenxmax, screenymax : \mathbf{N}$
$Pixel == 0..screenxmax \times 0..screenymax$
$Bit ::= black \mid white$
$PixelMap == Pixel \rightarrow Bit$
$PaintLayerIcon, ObjectLayerIcon : PixelMap$
$PaintTool, ObjectTool : \mathbf{P}\ PixelMap$
$selectpainttool : PaintTool$
attributes
$visible : PixelMap$
$images : \mathbf{P}\ PixelMap$
$vselected : \mathbf{P}\ PixelMap$
$layer : PixelMap$

tool : *PaintTool* ∪ *ObjectTool*
mousecursor : *PixelMap*

⋮

endagent *Visual*

Selection in both the paint and object layer are result predictable given the result templates we have chosen for *Paint* and *Object*. Restricting for the moment to the relationship between the agents *Paint*, *Visual*, and *Input* we can see that the attributes *visible* and *vselected*, which form a display template for the selection task, provide enough information for selection in the paint layer to be display predictable. They adequately portray the same information as the attributes *painting* and *pselected*, which form the result template. This assumes that we can adequately portray the mouse position information which identifies the region of points which will be sent as the argument for the *selectregion* message, and that is the function of the agent *PickPaint* in Figure 7.3.

For the object layer, it is obviously more difficult to ensnre display predictability since the display template has no means of portraying the hierarchical structure in the result template (it is slightly less difficult to portray the overlapping ambiguity, but still not trouble free). This problem could he overcome, and display predictability satisfied, if there was a way to present the structural hierarchy in the agent *Visual*. Translating the region of points determined by the monse into a set of object identifiers is performed by the agent *PickObjects* in Figure 7.4. For a user to be able to predict the outcome of selection, she must have access to as much information as *PickObjects* needs to perform that translation.

We can see that the paint layer is display predictable and the object layer is not. Since the ease with which complicated drawings can be manipulated is enhanced by the facilities given in the object layer, this lack of display predictability is not enough to abandon its use. However, the layering of paint and objects onto the same visual space makes matters even worse. In Figure 7.5 we present a graphical representation of the two layered SnperPaint agent description. There is no visible distinction between a painted image from the paint layer and a drawn image from the object layer. So not only does the user have to remember the grouping structure of the objects, she mnst also remember which images are objects and whicb ones are jnst paint. It may be possible to suggest a display strategy to circumvent this display unpredictability, bnt it seems unnecessary to make the distinction between the layers that has been forced hy the system description. Therefore, onr semiformal analysis of this system with respect to display predictability of the selection operator has uncovered a bad design decision.

In Chapter 4, we distinguished between predictability and consistency, since the latter was a generalization of the former in the simple agent model. In the refined model, the informal definition of consistency that we gave—the same input

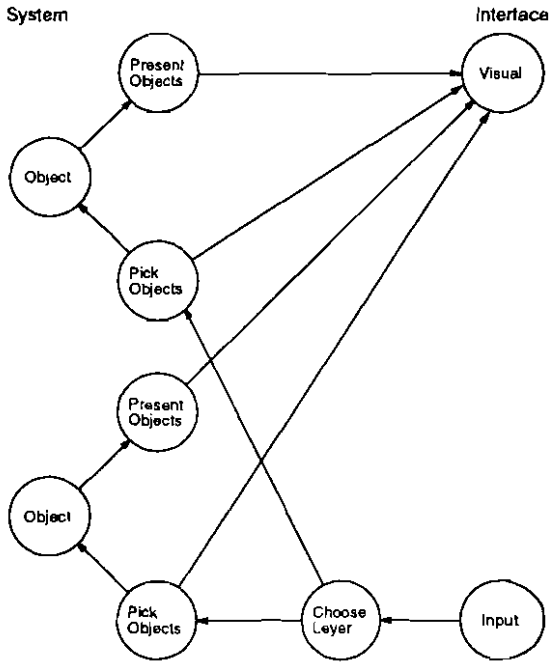

Figure 7.5: Agent diagram of the co-existing paint and object layers of SuperPaint

in similar situations has similar effects—is now very similar to predictability. Earlier, consistency was formalized in terms of a similarity criterion on the state set of an agent. Templates are a concrete way of formulating these similarity criteria. We noted earlier that definition of consistency did not capture task information. Similarity based on result and display templates does.

7.4 Synthesis

With the simple model of the agent, we stated that the observer of an agent only has access to the stimulus-response behaviour of the agent. Within the agent-based description of the *System* and *Interface*, this still applies. An agent can only participate synchronously in input or output events with another agent. Between the *User* and the computer, it is a bit artificial to name channels of communication

which serve as inputs to the user. Consequently, we have introduced display templates to represent the information from the computer which can be perceived by the user.

Synthesis describes the process by which the user determines how the effect of previous input on the result template for a particular task is observable via the display template for that task. The computer is honest with respect to a task if changes to the result template are immediately made apparent in the display template. If two histories are display equivalent, then they are also result equivalent. The computer is trustworthy if changes to the result template are eventually made apparent in the display template. Two histories must be display indistinguishable before they are considered result equivalent. Colloquially, honesty is paraphrased as the property:

> *If what you see now* (display template) *is the same, then what you have now* (result template) *is the same.*

whereas a trustworthiness conforms to the weaker property:

> *If all you can possibly see from now on is the same, then what you have now is the same.*

Formally, we would represent these task-centred properties as below.

$$
\begin{array}{|l}
Honest, Trustworthy : \mathbb{P}(TaskID) \\
\hline
task \in Honest \Leftrightarrow (\ \ task \in \mathrm{dom}\ Dtemplate \\
\qquad\qquad \wedge\ (\_ \approx_{dt} \_) \subseteq (\_ \approx_{rt} \_)) \\
task \in Trustworthy \Leftrightarrow (\ \ task \in \mathrm{dom}\ Dtemplate \\
\qquad\qquad \wedge\ (\_ \equiv_{dt} \_) \subseteq (\_ \approx_{rt} \_)) \\
\quad \textbf{where} \quad dt = Dtemplate(task) \\
\qquad\qquad \wedge\ rt = Rtemplate(task)
\end{array}
$$

We can give an example in the agent language which portrays the difference between honesty and trustworthiness, to show how honest interfaces can be more valuable. Our analysis will be conducted on a simplified version of a popular visual filesystem interface—the Macintosh desktop interface. We will expound on the 'New Folder' scenario briefly mentioned above and introduced by Roast and Wright [137]. Our description consists of a simple agent definition of a file system and a visual folder. The file system represents the *System* and the folder represents the *Interface*. Each description will only provide enough detail to discuss the scenario. We also will not make explicit the communication between the *System* and *Interface*, partly because that will be covered in the next chapter and partly because we need a definition of stable traces (also covered in the next chapter) to fully formalize the discussion.

The filesystem is a hierarchical arrangement of files and directories. Morgan and Sufrin [117] have provided a specification in Z of the Unix filing system, and our description is a simplified version of that. Files and directories are identified by elements in the set of all possible file identifiers, *FileID*. File identifiers are mapped to files in the system; we give no further description of files. The hierarchical arrangement is represented by two relations on file identifiers, one giving the unique *parent* of a file identifier and the other giving the set of *children* for a given file identifier. The distinction between files and directories is that directories can have children and files cannot. The hierarchy is acyclic and has a single *root* element, of which all other known files are descendants. At any time, one directory is *current*. We describe the filesystem as an agent below with only one operation, the creation of a new subdirectory under the current directory.

agent *Filesys*
internal
types
$[FileID, File]$
$root : FileID$
$emptydir : File$
attributes
$known : \mathbf{P}\ FileID$
$files : FileID \pfun File$
$parent : FileID \pfun FileID$
$child : FileID \rel FileID$
$currentdirectory : FileID$
$currentfiles : \mathbf{P}\ FileID$
invariant
$known = \mathrm{dom}\ files$
$root \in known$
$\mathrm{dom}\ parent = known - \{root\}$
$\mathrm{ran}\ parent \subseteq known$
$child = parent^{-1}$
$child^*(\!|\{root\}|\!) = known$
$currentdirectory \in known$
$currentfiles = child(\!|currentdirectory|\!)$
initially
$known = \{root\}$
operations
$\vdots$
$makesubdirectory(id : FileID)$
changes $(known, files, parent, child, currentfiles)$
pre $id \notin known$
post $files' = files \oplus id \mapsto emptydir$
$currentfiles' = currentfiles \cup \{id\}$

⋮

communication

inputs *filesysin* : *makesubdirectory*(*id* : *FileID*),...

outputs ...

r :...

external

...

endagent *Filesys*

The description of the *Interface* is given (partially) by the Desktop agent. The desktop contains folders, one of which is current. Each folder contains a set of icons, a subset of which are visible at any time (we only describe the visible icons for the current folder in the description below for simplicity). The current folder also displays the number of icons it contains. The only operation we define on the desktop agent is one to create a new folder icon in the current folder.

agent *VisualInterface*

internal

types

[*FolderID*, *Icon*]

desktop : *FolderID*

emptyfolder : *Icon*

attributes

folders : **P** *FolderID*

contents : *FolderID* ⇸ *Icon*

current : *FolderID*

visible : **P** *Icon*

invariant

folders = dom *contents*

current ∈ dom *contents*

desktop ∈ *folders*

visible ⊆ *contents*(*current*)

initially

folders = {*desktop*}

operations

⋮

newfolder(*fid* : *FolderID*)

changes (*folders*, *contents*, *visible*)

pre *fid* ∉ *folders*

post *folders*′ = *folders* ∪ {*fid*}

contents′ = *contents*⊕

current ↦ (*contents*(*current*) ∪ {*emptyfolder*})

visible ⊆ *visible*′

⋮

communication
inputs *folderin* : *newfolder*(*fid* : *FolderID*), ...
outputs ...
τ : ...
external
...
endagent *VisualInterface*

For the task of creating a new folder, the user wants to know if the current directory contains a new subdirectory, so the result template for this task is only the attribute *currentfiles*. When a new folder is created, this attribute value changes, increasing by one as the result of the inclusion of the (fresh) file identifier *id*. The display template corresponding to that result template is given by the attribute *visible*, indicating a subset of the icons which are contained in the current folder. However, the change due to the *newfolder* message does not require that the new icon be in the visible set. Macintosh users will be familiar with situations in which the new folder's icon does not appear in the immediately visible set of icons for the current folder. The system is not honest in this situation, requiring the user to browse through the current folder to observe that there is a new icon for the newly created folder. Hence, trustworthiness is all that can be claimed of this visual interface.

And we might add that we would expect all tasks to be trustworthy, so the visual interface is not gaining us much with this task over any other interface to a hierarchical file system. It might even be said to be worse than a command-based interface because the user of the visual interface is led to believe that if they don't see something change, then it hasn't changed. The visual interface is sly—it leads us to believe it is honest when it is not.

In defence of the Macintosh, there is a way to salvage honesty for this task. The visual interface provides another possible display template in the form of an *items* attribute which gives the count of the number of icons in the current folder. This display template honestly reflects the changes in the *currentfiles* result template. [2] We give the revised agent description of the visual interface below.

agent *VisualInterface*
internal
types
[*FolderID*, *Icon*]
desktop : *FolderID*
emptyfolder : *Icon*
attributes
folders : **P** *FolderID*

[2] Well, at least it tries to do so. It is possible to manipulate the window of the current folder in such a way that even this honest display template is obscured!

```
        contents : FolderID ⇸ Icon
        current : FolderID
        visible : P Icon
        items : N
    invariant
        folders = dom contents
        current ∈ dom contents
        desktop ∈ folders
        visible ⊆ contents(current)
        items = #(contents(current))
    initially
        folders = {desktop}
    operations
    ⋮
    newfolder(fid : FolderID)
        changes (folders, contents, visible, items)
        pre fid ∉ folders
        post folders' = folders ∪ {fid}
             contents' = contents⊕
                 current ↦ (contents(current) ∪ {emptyfolder})
             visible' ⊆ visible
    ⋮
communication
    inputs folderin : newfolder(fid : FolderID), ...
    outputs ...
    τ : ...
external
    ...
endagent VisualInterface
```

As Roast and Wright point out, even though this attribute maintains the honesty property, there is no guarantee that it is observed by the user, i.e., the *items* may not be used by the Macintosh user as a display template when creating new folder.

WYSIWYG

When the task is result predictable and honest, we satisfy the property:

> *What you can see now determines all you will be able to get.*

This sounds very much like the popular slogan "What you see is what you get" (WYSIWYG). A weaker, and consequently more realistic, property results from result predictability and trustworthiness and satisfies:

All you can possibly see determines all you will be able to get.

These two versions of WYSIWYG are summarized below.

$$
\begin{array}{|l}
WYSIWYG_{strong},\ WYSIWYG_{weak} : \mathbf{P}(TaskID) \\
\hline
task \in WYSIWYG_{strong} \Leftrightarrow (\ \ task \in \mathrm{dom}\ Dtemplate \\
\qquad\qquad\qquad\qquad \wedge\ (\_ \approx_{dt} \_) \subseteq (\_ \equiv_{rt} \_)) \\
task \in WYSIWYG_{weak} \Leftrightarrow (\ \ task \in \mathrm{dom}\ Dtemplate \\
\qquad\qquad\qquad\qquad \wedge\ (\_ \equiv_{dt} \_) \subseteq (\_ \equiv_{rt} \_)) \\
\quad \textbf{where} \quad dt = Dtemplate(task) \\
\qquad\qquad \wedge\ rt = Rtemplate(task)
\end{array}
$$

That these properties formalize the popular WYSIWYG slogan is slightly misleading because they only state that some desirable relationship holds between the result and display templates. Though this is a necessary condition for the interaction taking advantage of WYSIWYG, it is not sufficient since we do not guarantee that the relationship is understood by the user. But we are even more pessimistic than that. The formulation of this property does not even guarantee that the designer is aware of the relationship, because it does not have to be explicit in the agent description. In the next chapter, we will discuss localized correspoudence as a way to make explicit the relationship between result and display templates.

7.5 Result initiated interaction

The properties of the last section were all display initiated, that is, equivalence or indistinguishahility of display templates had implications toward equivalence or indistinguishability of result templates. Properties that are result initiated reverse the situation, so result information has implications toward display information. We do not have names for these properties. The first property satisfies:

What you can have (or know) now determines what you can see now.

and is characterized by the following implication, assumiug result and display templates rt and dt associated to the same task.

$$(\_ \approx_{rt} \_) \subseteq (\_ \approx_{dt} \_)$$

If the task is display predictable, we have the stronger implication satisfying:

What you can have (or know) now determines all you will be able to see.

and characterized by

$$(\_ \approx_{rt} \_) \subseteq (\_ \equiv_{dt} \_)$$

The weaker versions of the first property satisfies:

> *All you can possibly have (or know) from now on determines what you can see now.*

and is characterized by the followiug implication.

$$(\_ \equiv_{rt} \_) \subseteq (\_ \approx_{dt} \_)$$

When coupled with display predictability, we have the property satisfying:

> *All you can possibly have (or know) from now on determines all you will be able to see from now on.*

aud is characterized by the following implication, assuming result and display templates rt and dt associated to the same task.[3]

$$(\_ \equiv_{rt} \_) \subseteq (\_ \equiv_{dt} \_)$$

Since we have assumed that result templates bear significance to the task or knowledge structure of the user, it is possible that these last fonr properties would be considered more relevant within the user modelling domain.

7.6 Conclusions

In this chapter, we have investigated how the increased structure of the refined agent model allows the formulation of properties over collections of agents which are relevant to the internal goals of iuteraction—the results—and the tangible evidence of interaction—the displays. We have introduces templates as a task-centred means of restricting the knowledge of an agent's internal state. This provides us with the means of more clearly defining the scope of properties, confining them to the interactional unit of the task, iustead of over the computational unit of the agent, as was done in Chapter 4. Table 7.1 summarizes the task-dependent properties defined in this chapter.

Templates extend the bridge betweeu psychology and computer science, by forcing an iterative relationship between the psychologist and the interactive system designer. The HCI specialist provides the task analytic information, highlighting the tasks and the psychological attributes (or semautic features) related to each

[3]Sufrin and He classify a version of this last property as "goal determines view"

Result-Display relationship	Interactive Property
$(\_ \approx_{dt} \_) \subseteq (\_ \equiv_{dt} \_)$	display predictability
$(\_ \approx_{rt} \_) \subseteq (\_ \equiv_{rt} \_)$	result predictability
$(\_ \approx_{dt} \_) \subseteq (\_ \approx_{rt} \_)$	honesty
$(\_ \equiv_{dt} \_) \subseteq (\_ \approx_{rt} \_)$	trustworthiness
$(\_ \approx_{dt} \_) \subseteq (\_ \equiv_{rt} \_)$	WYSIWYG (weak)
$(\_ \equiv_{dt} \_) \subseteq (\_ \equiv_{rt} \_)$	WYSIWYG (strong)
$(\_ \approx_{rt} \_) \subseteq (\_ \approx_{dt} \_)$	result initiated interaction
$(\_ \approx_{rt} \_) \subseteq (\_ \equiv_{dt} \_)$	result initiated interaction
$(\_ \equiv_{rt} \_) \subseteq (\_ \approx_{dt} \_)$	result initiated interaction
$(\_ \equiv_{rt} \_) \subseteq (\_ \equiv_{dt} \_)$	result initiated interaction

Table 7.1: Task dependent interactive properties expressed as result-display relationships

task. The designer uses this information to formulate the result templates for the system. An agent description of the *System* can begin from that point. In designing the *Interface*, appropriate display templates must be determined for each result template in such a way as to maximize satisfaction of properties described above. If no such display templates can be found to satisfy even the weakest properties above, it may be the case that the task analysis was faulty, or incomplete, in which case some of the process can be iterated. Once some set of satisfactory display templates is settled, the agent description of the *Interface* the process is not complete as the display/result relationships must be subjected to tests to see if they make realistic demands on the user.

We realize that the relationship between display and result implied by the properties of this chapter must be made explicit in a design situation. In the next chapter, we will investigate how agents can be used to capture more explicitly the relationship between display and result.

Chapter 8

Interactive system architectures

Our purpose in applying formal methods to the analysis and design of interactive systems is to provide a means of expressing user recognizable structures in a language more closely related to the design practice. Abstract formalisms, such as the PIE model, and more concrete formalisms, such as the Sufrin and He model and the agent model of this thesis, have opened up the possibility of incorporating valid psychological assumptions of nsability as design principles in a more rigorous software engineering environment. The intention is that a principled design process will resnlt in the ability to engineer more usable systems. Currently, the majority of psychological knowledge concerning the usability of interactive systems has crept into design by means of common sense heuristics. We aim to show in this chapter how the agent formalism can capture the meaning behind some heuristics of interactive system architectnres.

Overview of chapter

In Section 8.1, we discuss the progression from abstract to concrete in interactive system design and how principles or properties that apply at the abstract level are mapped down to a concrete architecture. In Section 8.2, we describe two multiagent architectures for interactive system design, showing how they can he viewed in terms of the interaction framework. In Section 8.3, we discuss how the agent model allows a formal explanation and comparison of all levels from abstract to concrete, but especially at the concrete architectural level where few formalisms have been brought to bear.

In Section 8.4, we provide an example of how the agent model can formalize henristic properties of a multiagent architecture, specifically one role of the control component of a PAC agent. To do this, we concentrate on the abstract property of correspondence introduced in Section 4.3. We give a more constructive definition of correspondence with the aid of template information. Correspondence provides an architectural constraint on the relationship between collections of agents in the

Interface and *System*.

In Section 8.5, we investigate another architectural consideration in designing an interactive system. We use a semi-formal approach to justify how predictability and synthesis can be supported by the analysis of user recognizable structures in a graphical interface. The argument which we support in this extended example is that the agent architecture of the *System* specification should lead to a natural description of an *Interface* which will support the user's understanding of the system functionality. The particular case we examine concerns the possible arrangement of commands in the graphical interface of a multiple file text editor. It does not matter that our example is yet another specification of a text editor, because the scenario we describe concerns more the support of user recognizable structures in an interface that the text editing domain.

8.1 From abstract principles to concrete architectures

Thimbleby [160] introduced generative user-engineering principles, called *gueps*, as a means of capturing colloquially understood features of usability within the design process. A guep must satisfy the following four criteria.

- it can be expressed formally;
- it has a colloquial form that is accessible to users;
- it embodies valid psychological guidelines; and
- it is constructive, so that it indicates how the principle can be attained in design.

It is easiest to satisfy the first three criteria at an abstract level in which irrelevant clutter can be removed and concentration can be focussed on the essential features of the interaction. To address the fourth criterion, concrete architectural detail is needed. We identify three levels in the progression from abstract to concrete.

The conceptual level

The first and most abstract level identifies the context of an interactive system with a gross separation between the human and the computer. In this thesis, the interaction framework provides this conceptual and contextual description, and it has proved beneficial for satisfying the second and third criteria above. Measures applied to the translations within the framework gauge whether a component (*System*, *User*, *Input* or *Output*) can access the behaviour of another component and how easy it is to access that behaviour.

The layers level

At the second level, a finer grain of separation is introduced for both the human and the computer side of the interaction. At this level we introduce models such as Norman's seven stage model of the user activity and the Seeheim model of user interface management systems, both discussed in Chapter 3. Each of these models presents separation in terms of layers or stages of action. Norman's model certainly embodies valid psychological guidelines, since it was formed out of a psychological model of the user. The Seeheim model advocates the separation of presentation from application, which goes part of the way toward addressing the constructive criterion by isolating those features relevant to usability from those features relevant to functionality.

The architectural level

At the third and most concrete level, architectural models are introduced to directly address the construction of a system. In this thesis, we are interested in multiagent architectures, broadly defined as the class of architectures which advocates the identification of simple, independent computational units—agents—coordinated in order to produce complex behaviour. Multiagent architectures have several useful constructive features, as summarized by Coutaz [41]. They support iterative design since an agent defines the unit of modularity and can be altered without greatly affecting the behaviour of the rest of the agents in the system. They support distributed or parallel implementations. Considering the user as an agent which can communicate with system agents means that multithreaded behaviour, in which the user is participating in more than one task concurrently, can be more naturally expressed by assigning different agents to the different tasks of the user. In the next section, we describe two different multiagent models.

8.2 Multiagent models

Model-View-Controller

The Model-View-Controller (*MVC*) paradigm is the model used for the design of most Smalltalk interactive programs [107, 29, 96, 66]. On the surface, the MVC model fits very nicely with the interaction framework, as shown in Figure 8.1. An MVC triad represents three of the four components of the framework. The model is the object on which work in the application domain is to be performed, similar to the *System* in our framework. The controller provides the input interface to the user, similar to the *Input*. The view provides the output interface, similar to the *Output*. The user, though not explicit in MVC, is assumed to communicate directly with the controller and makes observations of the view.

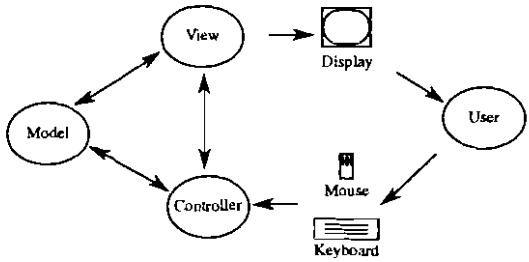

Figure 8.1: The Model-View-Controller paradigm of Smalltalk

At the abstract level of the interaction framework, we were not concerned with any further details on the decomposition of components. MVC, on the other hand, is implemented in an object-oriented language, thus providing a natural mechanism for a hierarchical arrangement of MVC triads. Figure 8.2 shows a typical hierarchical arrangement of a collection of windowed interactive applications. Another distinction between the interaction framework and MVC can be seen by the explicit communication link between controller and view; there is no such explicit connection between *Input* and *Output* in the framework. This is another difference that arises from the different levels of abstraction they are intended to address.

Presentation Abstraction Control

Coutaz has suggested an alternative to the MVC paradigm, called the Presentation Abstraction Control—or *PAC*—model, shown in Figure 8.3 [38, 39, 40]. The abstraction component corresponds to the model of MVC and the *System* of the framework. The presentation component combines both view and controller of MVC, similar to the *Interface* of the framework. In MVC, the view and controller communicate directly with the model, and vice versa. In PAC, this communication is coordinated by the control component to ensure that the state of the presentation component faithfully portrays the state of the abstraction component. The control component also coordinates communication between separate PAC agents hierarchically arranged.

PAC is intended as a high-level design notation for the description of an interface. It does not have an underlying implementation as MVC does. However, PAC is still intended to express a hierarchical relationship between a collection of interface objects. Figure 8.4 gives the relationship between a PAC hierarchy and the other components of an interactive system. The PAC hierarchy can be seen as the dialogue and presentation control.

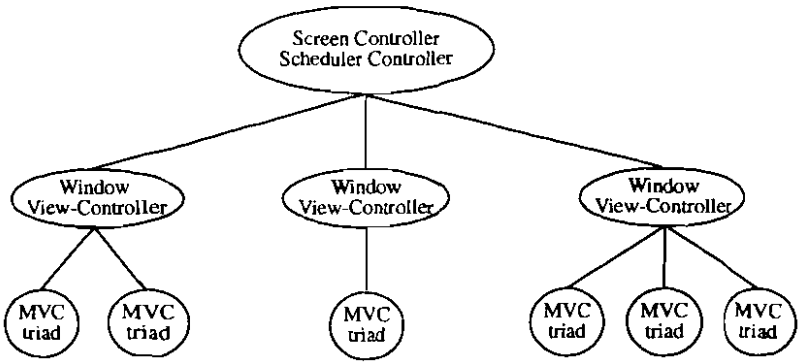

Figure 8.2: The MVC hierarchy

8.3 Applying formal methods to levels of abstraction

Having described the three levels progressing from abstract to concrete descriptions of an interactive system, we need to show how properties which arise at the most abstract level are traced down to the concrete, architectural level. We have adopted the ageut as the formal means for propagating interactive properties. At each level described above, the inspiration for the properties to be formalized comes from common sense heuristics. Formality, therefore, does not replace intuition; the two are complementary.

At the most abstract level, there are computational formal models, such as the PIE model in which to express many gueps of interest. The agent model can express the same properties as the PIE model by looking at its trace-state behaviour. An example of an abstract psychological model is Norman's execution and evaluation model, which was the primary motivation for tbe interaction framework and the simple agent model.

At the next level, where there is more separation between components, the formal models need to express relationships between the layers or stages. The state-

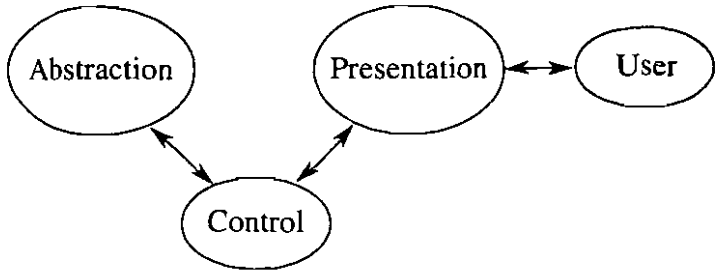

Figure 8.3: The PAC model

display model for the system and CCT for the user are examples of such layered formal models. There is also a need to relate properties which are expressed on the layered models to the abstract properties given by the PIE model, the agent model and the interaction framework. Agents can represent the layers or stages—for example, the state-display model becomes the *System-Interface* model—and so properties expressed as constraints between agents apply to this level.

At the architectural level, there are no existing examples of a formal model which captures interactive properties. The Forest project has produced an agent-based description of the MVC model within a modal action logic (MAL) [66, 141], but the purpose of that formal model was not to investigate the interactive properties of MVC. This absence of formalism at the constructive stage of interactive system design was one of the prime motivations for the refined agent model. One of the problems with the multiagent models described above is that while they do address very strong heuristic notions of usability—for example, catering for multithreaded dialogue between user and system so that the user is freed to direct the interaction—the lack of a formal model does not allow the description of desirable properties such as predictability, consistency, honesty, etc. that were valuable at the more abstract levels. It is also difficult to compare different multiagent architectures when they are expressed informally. The refined agent model allows us to formalize features of a multiagent architecture, for a more honest comparative appraisal.

The first step formalizing a multiagent architecture is to reject the fundamental units of both MVC and PAC. MVC suggests that the model-view-controller triad is the basis for the fundamental unit of interactive system design. PAC suggests the same for the PAC agent. In practice, however, this informal rule is continually broken, to the extent that the separations implied by each model are too artificial to be useful. View-controller pairs without an associated model are common in Smalltalk interactive systems in order to affect flow of control (for example, to express the role of a parent window which administers the change of control between

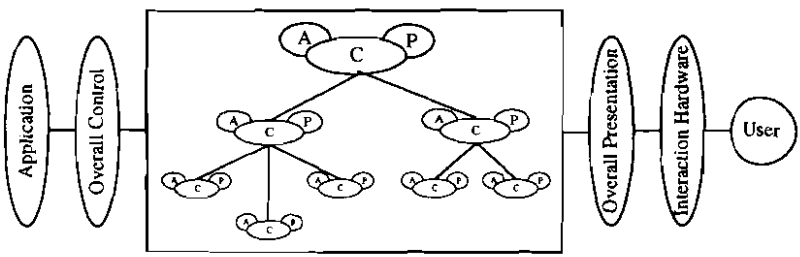

Figure 8.4: The PAC hierarchy within an interactive system

subwindows [66]). Similarly, in examples of PAC diagrams, it is not uncommon to see PAC agents without a presentation or an abstraction component, or with multiple presentation components. More recent models of PAC by Nigay [121] allow communication between two separate application components to bypass the control components. While we may be surprised by such glaring inconsistencies, it is not surprising that they arise. The PAC and MVC architectures are intended as a guides for designers; they are not intended to be strictly enforced as would be demanded by a formal model.

The agent, as defined in this thesis, is a well-defined building block for interactive system design, unlike the MVC triad or the PAC compound agent.[1] We can now view the MVC and PAC models for what they are—heuristic guidelines for the construction of agents into interactive objects. The MVC triad and the PAC compound agent can be realized as composed agents which can address one or more of the user's tasks. The model or abstraction represents result information. The view-controller pair or the presentation represents display information.

What was not discussed in Chapter 7 was how the relationship between the

[1]We use the term compound agent now to distinguish the PAC unit from our formal agent model.

model and its view or the abstraction and its presentation is to be maintained. Herein lies a major difference between the MVC and PAC models. Correspondence between model (respectively, abstraction) and view (respectively, presentation) must be maintained in MVC (respectively, PAC) in order to support the user's understanding of the result of interaction based on observations of the display. In PAC, the control component makes the correspondence explicit and separate from either presentation or abstraction. In MVC, the correspondence must be maintained either within the model or the view, or both, meaning that model and view are not as separable as may be desired.

But how does the control component of a PAC agent maintain the correspondence? In the following section, we will develop the formal notion of correspondence introduced in Chapter 4 in an attempt to formalize the role of the control.

8.4 Local correspondence

Recall in the definition of correspondence from Section 4.3 that two agents are said to correspond if the operations associated with input for one is sufficiently mirrored in the other according to some *retrieve* relation between their state sets. We need to slightly revise the definition of correspondence to relate to the refined agent model. In order for agent B to correspond to agent A, we only require that retrieve relation hold for *stable* traces of the two. A stable trace is one which can only be extended by a stimulus event, that is, the agent can only proceed by participating in some input event.

$$
\begin{array}{|l}
stable : Agent \rightarrow \mathbf{P}(\mathrm{seq}\ Event) \\
\hline
t \in stable(A) \Leftrightarrow (\ \ t \in \mathcal{H}[\![\ A\ ]\!] \\
\qquad\qquad \wedge \forall t' : \mathcal{H}[\![\ A\ ]\!] \\
\qquad\qquad\quad \mid\ t \frown t' \wedge t \neq t' \\
\qquad\qquad\quad \bullet\ t'(\#t + 1) \in stimuli(A)
\end{array}
$$

The definition for correspondence for the refined agent model is very similar to the one given in Section 4.3 with the addition of stability conditions.

$$
\begin{array}{|l}
\textit{AgentCorr} \\
\hline
\textit{Agent}^I \\
\textit{Agent}^J \\
\textit{retrieve} : \textit{states}^J \leftrightarrow \textit{states}^I \\
\hline
\textit{inits}^J \times \textit{inits}^I \subseteq \textit{retrieve} \\
\\
\forall (tr_I, s_I), (tr_I \frown tr'_I, s'_I) : I^{int}_{\theta \textit{Agent}^I} \\
\quad | \quad (\quad (s_I, s'_I) \in (tr'_I \,\mathbin{⨾}\, \mathrm{mess} \,\mathbin{⨾}\, \textit{operations}^I) \\
\qquad \land \{tr_I, tr_I \frown tr'_I\} \subseteq \textit{stable}(\theta \textit{Agent}^I) \\
\quad \bullet \quad \exists (tr_J, s_J), (tr_J \frown tr'_J, s'_J) : I^{int}_{\theta \textit{Agent}^J} \\
\qquad | \quad (\quad (s_J, s'_J) \in (tr'_J \,\mathbin{⨾}\, \mathrm{mess} \,\mathbin{⨾}\, \textit{operations}^J) \\
\qquad\quad \land \{tr_J, tr_J \frown tr'_J\} \subseteq \textit{stable}(\theta \textit{Agent}^J) \\
\qquad \bullet \quad \{(s_I, s_J), (s'_I, s'_J)\} \subseteq \textit{retrieve}
\end{array}
$$

We define $correspond\_$ as a relation between agents indexed by a retrieving relation between their state sets. The pair of agents (D, R) is in $correspond_{ret}$ if agent R corresponds to agent D as defined by the schema *AgentCorr* with retrieving relation ret.

$$
\begin{array}{|l}
correspond\_ : (\textit{State} \leftrightarrow \textit{State}) \rightarrow (\textit{Agent} \leftrightarrow \textit{Agent}) \\
\hline
correspond_{ret} = \{ \ \textit{AgentCorr} \\
\qquad\qquad\qquad | \quad \textit{retrieve} = ret \\
\qquad\qquad\qquad \bullet \quad \theta \textit{Agent}^I \mapsto \theta \textit{Agent}^J \ \}
\end{array}
$$

Within the interaction framework, we want the *User*'s interaction with the *Interface* to be so mirrored by the *System*. Overall, we would want to satisfy the following constraint.

$$
\begin{array}{l}
\exists \, \textit{AgentCorr} \\
\bullet \quad (\quad \theta \textit{Agent}^I = \theta \textit{Interface} \\
\qquad \land \, \theta \textit{Agent}^J = \theta \textit{System})
\end{array}
$$

This means that we have to find the *retrieve* relation to interpret information of the *System* within the *Interface*. Motivated by the multiagent architectures, it is desirable to localize the correspondence between *Interface* and *System*, both to make it easier to satisfy the demands of correspondence and to reflect the relationship between interactive objects as seen by the user and their counterparts within the *System*. The display and result templates introduced in Chapter 7 are useful for such localization.

$$[\textit{TaskID}]$$

$$
\begin{array}{|l}
Rtemplate : TaskID \pfun Template \\
Dtemplate : TaskID \pfun Template \\
\hline
\mathrm{dom}\, Rtemplate = \mathrm{dom}\, Dtemplate \\
Rtemplate(t) \subseteq System.attribs \\
Dtemplate(t) \subseteq Interface.attribs
\end{array}
$$

This task information forms the basis for the local correspondence between *System* and *Interface*. For every task, we require a retrieve relation between the values of the result attributes and the display attributes for that task.

$$
\begin{array}{|l}
taskretrieve : TaskID \pfun (State \rel State) \\
\hline
\mathrm{dom}\, taskretrieve = \mathrm{dom}\, Rtemplate \\
\mathrm{dom}(taskretrieve(t)) \subseteq \{\ s : System.states \bullet Rtemplate(t) \rhd s\ \} \\
\mathrm{ran}(taskretrieve(t)) \subseteq \{\ s : Interface.states \bullet Dtemplate(t) \rhd s\ \}
\end{array}
$$

Given the result and display templates for the interactive system, we can then require that for any task identifier t, the *System* restricted by the result template for t corresponds to the *Interface* restricted to the display template for t. Below we give a formal representation of this local correspondence.

LocalCorrespondence

$$
\begin{array}{|l}
System, Input, Output, Interface : Agent \\
tasks : \mathbb{P}\, TaskID \\
Rtemplate : tasks \pfun Template \\
Dtemplate : tasks \pfun Template \\
taskretrieve : tasks \pfun (State \rel State) \\
\hline
Interface = compose_{sync}(Input, Output) \\
Rtemplate(t) \subseteq System.attribs \\
Dtemplate(t) \subseteq Interface.attribs \\
\mathrm{dom}(taskretrieve(t)) \subseteq \{\ s : System.states \bullet Rtemplate(t) \rhd s\ \} \\
\mathrm{ran}(taskretrieve(t)) \subseteq \{\ s : Interface.states \bullet Dtemplate(t) \rhd s\ \} \\
\forall\, t : tasks;\ rt : Rtemplate(t);\ dt : Dtemplate(t) \\
\quad \bullet\ (System \Downarrow rt, Interface \Downarrow dt) \in correspond_{taskretrieve(t)}
\end{array}
$$

The PAC compound agent is related to a set of tasks for the user. Given a task, the description of the abstraction agent supports that task in terms of the attributes in the result template. Related attributes for the presentation agent are

then chosen and their values are constrained by a *retrieve* relation. The behaviour of the control agent is in part specified by the local correspondence relationship that it must maintain for each task the PAC compound agent is meant to support. We say in part because the control component also coordinates communication between PAC multiagents and not just between its presentation and abstraction agents.

The *System* is derived from a PAC description as the synchronized composition of all of the abstraction agents. The *Interface* is derived as the synchronized composition of all of the presentation agents. *Perform* and *Present*, the agent manifestations of the *performance* and *presentation* translations are derived from the synchronized composition of all of the control agents. We can further divide the control agent into a performance subagent controlling dialogue flow from presentation to abstraction, a presentation subagent, controlling dialogue flow from abstraction to presentation, and a hierarchical subagent, controlling dialogue flow between other control agents.

In the next section, we will concentrate on an example showing how local correspondence can be used to suggest a natural graphical interface to a *System* description.

8.5 Assessing the graphical interface to a text editor

In this section, we use the agent language to motivate a semi-formal support for predictability and synthesis. The detail of strict formality sometimes clouds the insight which it can support, and so seek in this example to demonstrate how the formal properties of predictability and synthesis can be used to formulate heuristic guidelines for architectural constraints for agents.

From the description of the interaction framework, we know that the presentation of the interface has great impact on the effectiveness of the interaction. Given an agent description of the *System*, we can provide a realistic example of how to increase the effectiveness of the graphical interface. The example we use extends from work done on the ESPRIT Basic Research Action project 3066 (AMODEUS) [13, 118] involving scenarios of interestiug user behaviour within an interactive system. Within the project, scenarios are nsed to compare and contrast the effectiveness of system and user modellers iu explaining interactive phenomena. The scenario we will describe concerns the grouping of commands in two versions of the graphical interface to a mnltibuffer text editor. The versions of the editor are Spy (version 9) and its direct descendant Ten (presumably short for Spy version 10). Both were developed at the Rutherford Appleton Laboratories in the United Kingdom.

In this example, the snippet of action involves the invocation of operations

performed on selected text. The advantage of a multiple file, or multibuffer, text editor is that the user can rapidly switch attention between different files. Changes to the text of one file which involve text from another file—for example, the copying of some text from one file to a specified place in the other—are readily provided for in a multibuffer editor. In this example, the nser may have many buffers active at once, each containing text from some file in the filesystem, but only one of those buffers is the current buffer—the one to which active editing commands such as insertion of a character are directed. Also, the user may select a contiguous regiou of text within the current buffer. This selected text may be deleted from the buffer, or it may be copied or moved to any of the open buffers in the editor.

The multibuffer editors on which this scenario is based have many other features, most of which are deserving of further analysis, but for the purposes of this scenario, the functionality of the operations to be performed on the selected text are all that concern us. The challenge posed to HCI modellers is to assess two options for the arrangement in the graphical interface of the operations for deletion, copying and moving of selected text.

The first option is represented in the editor Spy (version 9), and is shown in Figure 8.5. In this graphical interface, the operations to copy and move are grouped together on a pop-up menu and the delete operation is separated and appears on the menu bar associated to each buffer. The second option is represented in the editor Ten, and is shown in Fignre 8.6. In this graphical interface, all of the operations—delete, copy and move—are grouped together on the same pop-up menu.

We begin with an agent description of the *System* component of the multibuffer editor. The specification of the functionality of the multibuffer editors represented by Spy and Ten is a reverse-engineered description. A separate specification of Spy has been presented by Martins exclusively in the Z notation [106]. We hope to demonstrate that the agent language provides an equally powerful specification technique with the advantage that it more closely reflects the object-oriented view crucial to the relationship between the *System* and *Interface*.

After presenting a specification of the *System* component of the multibuffer editor, we will move on to discuss the graphical interface. Our assumption again is that both Spy and Ten share a common *System* descriptiou—at least as far as this scenario is concerned—but differ in the arrangement of the graphical interface. The *System* description leads us to the suggestion of a natural graphical presentation of the *System* which we could also be described in terms of agents. This natural graphical interface can then be compared to the actual interfaces for both Spy and Ten.

We use the agent language to give a limited descriptiou of the functionality of the multibuffer editor. Our agent description is limited because we will only define that part of the editor that concerns the scenario described above, i.e., we will only describe enough to facilitate specification of the operations of deletion, copying and

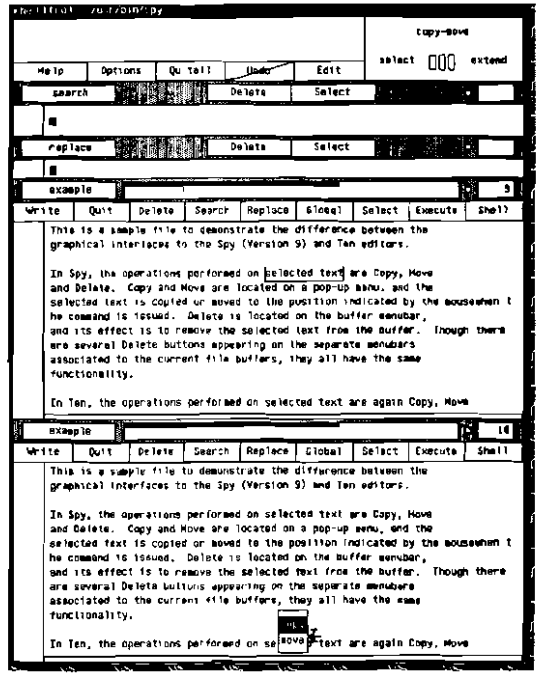

Figure 8.5: The graphical interface of Spy

moving of selected text.

Figure 8.7 is a simple graphical representation of an agent architecture for the multibuffer editor functional core. In Figure 8.7, we represent the *Interface* as a single agent, and the relationship between interface and functional core is represented by the *Perform* and *Present* agents.

A single buffer will contain text and we will describe operations that manipulate text both within and between buffers. Within a buffer, the usual insertion and deletion of single characters is allowed at an insertion point which can be set to any point within the text of the buffer. A section of text within a buffer can be selected for subsequent deletion or it can be copied or moved to any point within the text of any buffer. There is only one selected section of text in the whole multibuffer editor at any one time.

We will now give descriptions for the *System* agents in Figure 8.7.

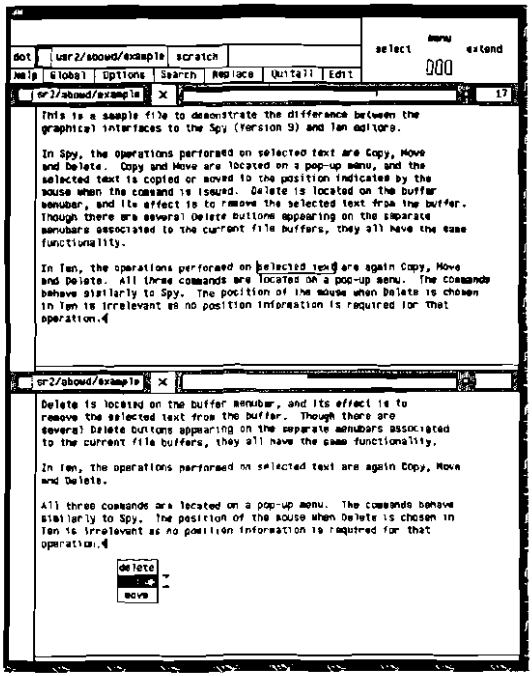

Figure 8.6: The graphical interface of Ten

8.5.1 Single buffer

Figure 8.8 is a graphical representation of the single buffer agent. We begin the description of the single buffer by describing its state and how it is initialized. For the purposes of this description, the only attributes of interest involve the main text and the selected text within the main text. We are making the explicit assumptions about the the result template for these editing tasks, and are defining the *System* as restricted by this result template. Text is a sequence of characters, chosen from a set of all possible characters (CH). The selected text, if it exists, is delimited by two natural number indices within the main text. If there is no selected text, i.e., it is of value *null*, then these indices also have the value *null*. A non-*null* insertion point indicates where subsequent characters are inserted into the text. The insertion point is an index within the text that splits the text into two subsequences---the text before the insertion point and the text after the insertion point. Initially, the contents of the buffer is empty. The agent description for this core of the buffer is

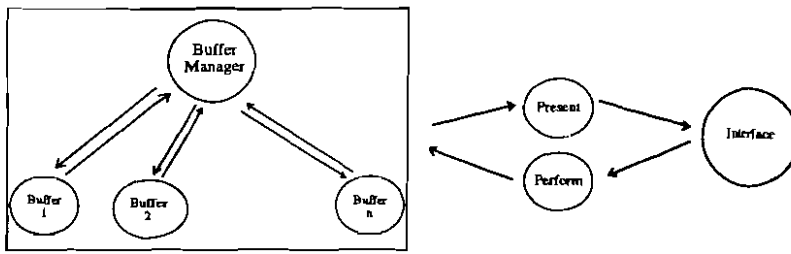

Figure 8.7: Agent representation of multibuffer editor

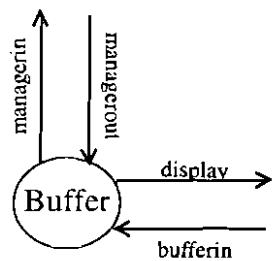

Figure 8.8: The single buffer agent

given below.

agent *bufferstate*
internal
types
$[CH]$
$\blacksquare \notin CH$
$null \notin CH$
$Text == \text{seq}(CH \cup \blacksquare)$
$\forall t : Text \bullet (\quad last(t) = \blacksquare$
$\wedge \blacksquare \notin \text{ran}(front(t))$
attributes
$text : Text$
$before, selected : \text{seq}\ CH \cup \{null\}$
$insertion, begsel, endsel : \mathbb{N} \cup \{null\}$
invariant

$$
\begin{aligned}
&\mathit{insertion} = \mathit{null} \Rightarrow (\ \ \mathit{begsel} = \mathit{null} \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{endsel} = \mathit{null} \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{selected} = \mathit{null} \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{before} = \mathit{null}) \\
&\mathit{insertion} \neq \mathit{null} \Rightarrow (\ \ \mathit{insertion} \in \mathrm{dom}\ \mathit{text} \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{before} = 1..(\mathit{insertion} - 1) \lhd \mathit{text} \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{begsel} = \mathit{insertion} \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{endsel} \geq \mathit{begsel} \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{selceted} = \mathit{begsel}..\mathit{endsel} - 1 \overset{\mathrm{seq}}{\lhd} \mathit{text})
\end{aligned}
$$

initially

$\mathit{text} = \langle\ \rangle$

$\mathit{insertion} = \mathit{null}$

operations

communication

external

endagent *bufferstate*

Text in the buffer is terminated by the special end of text symbol ■, which is not in the set CH. The state invariant embodies the link between the insertion point and the selected text. If there is no insertion point for a buffer, then there is no selected text. In addition, we constrain the selected text to occur as a contiguous sequence of text positioned directly after the insertion point.

We can now specify the operations which deal with selection of text. These operations will be collectively specified in the agent *bufferSelect*. The first two operations are marking the beginning of the selection of text and extending the selection of text. These are intended to be initiated by the *User*, via the *Interface*. They are specified in the agent *bufferMarkSelect*. After either a mark or extend event, the buffer manager is informed of the new selected text.

agent *bufferMarkSeleet*
interleaves (*bufferstate*)
with
internal
operations
beginmark($n : \mathbb{N}$)
changes (*before*, *selected*, *insertion*, *begsel*, *endsel*)
pre
$n \in \mathrm{dom}\ \mathit{text}$
post
$\mathit{insertion}' = n$
$\mathit{begsel}' = \mathit{endsel}'$
extendmark($n : \mathbb{N}$)
changes (*before*, *selected*, *insertion*, *begsel*, *endsel*)
pre

$n \in \mathrm{dom}\ text$
$insertion \neq null$
post
$n \leq begsel \Rightarrow (\ begsel' = n \wedge endsel' = endsel)$
$n > begsel \Rightarrow (\ begsel' = begsel \wedge endsel' = n)$
selecttext$(t : \mathrm{seq}\ CH)$
changes ()
pre
$t = selected$
communication
inputs
bufferin : *beginmark*$(n : \mathbb{N})$; *extendmark*$(n : \mathbb{N})$
outputs
managerin : *selecttext*$(t : \mathrm{seq}\ CH)$
external
$\mu X \bullet (bufferin?x \mathbin{--} managerin!selecttext(t) \rightarrow X)$
endagent *bufferMarkSelect*

Notice how the external description of *bufferMarkSelect* is so structured to make the stimulus-response of this agent clear. We also note that this external description would allow extending the selection before the selection has begun. You can do a (*bufferin*, *extendmark*(n)) before a (*bufferin*, *mark*(n)) has been done, but this is disallowed by the internal description of the agent operations (specifically, the precondition of the *extendmark* operation prevents this).

The next operation, unselecting text, is a control operation which will be requested by the buffer manager to ensure that only one of the buffers contains selected text. Its isolated specification is given by the agent *bufferUnselect*.

agent *bufferUnselect*
interleaves (*bufferstate*)
with
internal
operations
unselect()
changes (*before*, *selected*, *insertion*, *begsel*, *endsel*)
pre
$insertion \neq null$
post
$insertion' = null$
communication
inputs
managerout : *unselect*()
external

```
    run
endagent bufferUnselect
```

The final operations at the buffer level describe the insertion of text at an arbitrary position (used for copying and moving) and the deletion of the selected text.

```
agent bufferInsDel
interleaves (bufferstate)
with
internal
    operations
    insert(t : seq CH; n : N)
        changes (before, selected, insertion, begsel, endsel, text)
        pre
            insertion ≠ null
        post
            insertion' = n
            before' = 1..(n − 1) ◁ text
            endsel'..#text' ◁^seq text' = n..#text ◁^seq text
            selected' = t
    remove()
        changes (selected, endsel, text)
        pre
            insertion ≠ null
        post
            selected' = ( )
            endsel'..#text' ◁^seq text' = endsel..#text ◁^seq text
communication
    inputs
        managerout : insert(t : seq CH; n : N); remove()
external
    run
endagent bufferInsDel
```

We can finally combine the above three agents to arrive at a specification of a single buffer agent with selection facilties, which we call bufferSelect. We add the display channel at this point for communication to the *Interface*. The implicit external constraint states that every input event to the buffer is followed by an output event to the display.

```
agent bufferSeleet
interleaves (bufferMarkSelect, bufferUnselect, bufferInsDel)
with
internal
```

operations
showtext(t : seq CH; m, n : $\mathbb{N} \cup null$)
pre
$t = text$
$m = begsel$
$n = endsel$
communication
outputs
display : *showtext*(t : seq CH; m, n : $\mathbb{N} \cup null$)
external
satisfying
$\forall t$: *traces*
• $\#(t \mid \{display\}) \leq \#(t \mid \{bufferin, managerout\}) \leq \#(t \mid \{display\}) + 1$
endagent *bufferSelect*

We could continue to define further behaviour of a single buffer, but for the example of this scenario, the behaviour of *bufferSelect* is adequate, and so we can refer to it as the *buffer* agent.

agent *buffer*
interleaves (*bufferSelect*)
endagent *buffer*

8.5.2 The buffer manager

The concurrent activity of a collection of buffers will be managed by the agent *manager*. The purpose of this agent is twofold. It will ensure that only one buffer is active, that is, only one buffer contains a non-null insertion point. It will also provide a means by which the selected text is deleted, copied, or moved. We will need a set of buffer labels, $BUFFID$, which will serve as unique identifiers for all of the possible buffers that could be known to the manager. The manager is initialized internally.

agent *managerState*
internal
types
$BUFFID == \mathbb{P}\, \mathcal{V}$
$null \notin BUFFID$
attributes
known : $\mathbb{P}\, BUFFID$
current, *last* : $BUFFID \cup \{null\}$
selected : seq CH
invariant
$current \neq null \Rightarrow current \in known$
$last \neq null \Rightarrow last \in known$

initially
known = ∅
communication
external
endagent *managerState*

We will not give the detailed description of the buffer manager at this point. The importance of this description is that it would contain events corresponding to the copying and moving of selected text from one buffer to another. Input from the *User* via the interface that is intended to perform these two operations on selected text must be controlled via the manager; interaction directly with the buffer would not be able to control the communication of selected text between buffers. In fact, it is precisely because individual buffer agents cannot control this communication that the buffer manager was introduced. The manager ensures that only one buffer is active at a time and it keeps track of the selected text in order to communicate it to any buffer necessary.

It is not necessary for the buffer manager to control deletion of the selected text, since that operation can be defined at the buffer level. However, a design decision for these text editors has placed the delete command at the control of the buffer manager.

8.5.3 Deriving a description of the interface

As we suggested earlier, the agent description of the functional core suggests a natural agent description of its graphical interface. Figure 8.9 is a more detailed agent representation of the functional core. This figure shows how operations relating to selected text are distributed in the agent hierarchy. The copy, move and delete commands are located within the buffer manager, and the other operations on the selected text—marking the beginning and extending the selected region—are located at the buffer level.

Predictability, as described earlier in this thesis, centres around the future results of the interaction being determinable based on knowledge of what the results are currently, and likewise for the displays. Synthesis concerned whether the user was able to determine changes to the result based on observed changes to the display. If the result and display behaviour is predictable, then to satisfy properties associated to synthesis (honesty and trustworthiness are the ones we have discussed explicitly in this thesis), we need to ensure a correspondence between the structures the user recognizes from the graphical interface and the result structures the designer intended the display to portray.

With this in mind, we suggest that the agent description depicted in Figure 8.9, and given formally earlier, leads to a natural description of the graphical interface which corresponds to the agent description of the *System*. We can contrast this with

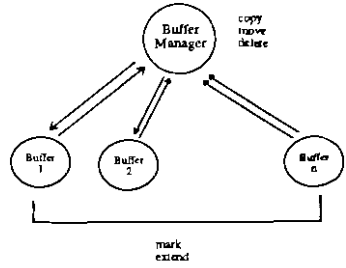

Figure 8.9: Agent representation of functional core with operations.

an agent description of the interfaces to Spy and Ten based on the visual appearance of the graphical interfaces shown in Figures 8.5 and 8.6. In Figure 8.10, we give the agent description of the Spy interface. Here it is seen that the delete operation

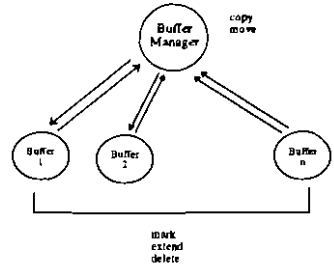

Figure 8.10: Agent representation of Spy's interface.

is distributed to the presentation of each buffer agent. User of this system will falsely connect the functionality of the delete selected text operation to the buffer containing the selected text. But experimental interaction with the interface shows that pressing *any* delete button will always delete the selected text.

In Ten (see Figure 8.11), since the delete command has been included in the pop-up menu with copy and move, it is no longer falsely associated to the single buffer. We say this because the pop-up menu is not connected graphically to any individual buffer, but rather roams independently with the mouse, whose input to the system is constrained to the region in which the editor is located.

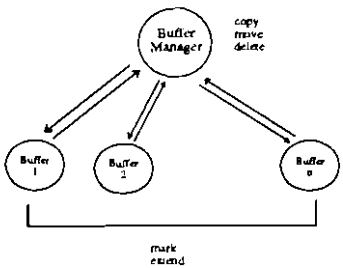

Figure 8.11: Agent representation of Ten's interface.

8.5.4 Conclusions on the case study

Though the graphical interface of Ten corresponds more closely to the functional description of the text editor when restricted to the tasks of deleting, copying and moving selected text, there still remain questions about its implementation. By placing the delete operation on the pop-up menu, deletion was removed from its ill-advised connection to the individual buffer. However, pop-up menus attached to the mouse are linked with the positional information associated to the monse as well. Whereas this information is necessary for the copy and move operations, because their operations depend on positional information (the destination of the moved or copied text), it is not necessary for deletion, which is an operation independent of the monse position. This argument leads us to suggest that the best location for delete would have been in some overall text editor menu, or even attached to some key on the keyboard. Argnments for placing deletion on the pop-up menu centre around minimizing the motor movement and distraction of the user. Since deletion will often take place shortly after the mouse has been used to mark and extend the selected text, it would be easiest and least distracting to the user to allow them to use the mouse again to issue the deletion command. We cannot sufficiently support these kinds of keystroke level or attention arguments within our formal interaction models. As a result, we admit that our formal techniques can only truly be effective in concert with other psychological modelling techniques which will provide decision support for design where our models cannot.

There are several reasons why this particular multibuffer text editor provides a very interesting case study. For the purposes of the scenario, the two versions are functionally equivalent. Their only difference lies in the arrangement of commands in their graphical interfaces. There is a significant difference in the two graphical interfaces, and one must ask the reason for the massive changes. Ultimately, tbe changes were made in order to increase the amount of screen real estate dedicated

to the display of the buffer contents.[2] Our analysis has tried to trace the usability conseqnences of this design decision.

It tnrns out for this particular of command organization, the later version (Ten) was an improvement ou usability. Other analysis that we have done [97] concerning the search/replace facility shows that Teu is indeed less usable because the correspondence between result and display templates for the searching task is not as the user anticipates. In general, thc move to greater screen real estate has meant a sacrifice in visibility of correspondence detail that facilitates predictability.

8.6 Conclusions

In this chapter, we have initiated an assessment of multiagent architectures with the formal agent model. Having introduced two well-known multiagent architectures, MVC and PAC, we identified a major difference between them iu terms of the abstract property of correspondence which we desire to map down to a concrete form in some architecture. PAC advocates the isolation of correspondence information between *System* and *Interface* agents, whereas MVC does not. We presented local correspondence as a more task-related form of the abstract version of correspondence presented in Chapter 7. The formal definition led directly to a more precise interpretation of the duties of the control agent in a PAC compound agent, which we isolated in terms of its correspondeuce role and its hierarchical role.

Iu the case study of Spy vs. Ten, we showed how a formal development of the *System* agent architecture can lead to a semi-formal heuristic for the design of potential graphical interfaces to support properties such as predictability and synthesis, in which the user tries to recognize structures of the *Interface*and relate them to structures of the *System*.

[2] This information was obtained through private communication with David Duce from Rutherford Labs.

Chapter 9

Conclusions

9.1 Summary of the thesis

Recalling the dual focus of attention in this thesis—human-computer interaction and software engineering—we can summarize the results which have been presented.

From an HCI perspective, we have provided informal, semi-formal and fully formal mechanisms for the design and analysis of interactive systems. Informally, we have defined an interaction framework which describes the major components in an interactive system, the user, the system and the interface, and we used this framework both to contextnalize a great portion of HCI research and to motivate the more formal treatment of interactive properties. At the more formal level, we have provided a constructive bridge between the abstract and general computational models of interaction and the heuristic design of interactive systems. A formal model of the agent has been nsed as the building block for interactive system description. We have provided the beginnings of a methodology for incorporating psychological knowledge of users by demonstrating the link between task analysis and agent description. The formulation of interactive properties on agents provides constraints for .the design of an interactive system and a means for evaluating existing systems to understand user difficulties. We have provided several examples of how a formal or semi-formal analysis can be nsed to explain scenarios of user hehaviour.

From a software engineering perspective, the agent model is a formal model of an object-oriented specification language. It is the first specification language to our knowledge with a compositional semantics that incorporates existing model-oriented techniques for a description of an entity's internal state behaviour along with existing process algebra techniques for a description of the external event hehavionr. The agent language answers criticisms of previous model-oriented techniques, such as Z, by providing a mechanism for describing objects. Beyond that, we have also incorporated a means for specifying the communication between objects,

a facility which is not treated as generally in other object-oriented formalisms.

It is interesting to note that the requirements for our agent model which lead to the development of the hybrid notation arose directly from our need to express a particular class of interactive properties. Thus we have shown how due consideration of nonfunctional requirements in software engineering can increase the benefits of formal methods as well as influence the development of better formal techniques.

9.2 Contributions of this thesis

Our goal in this thesis was to provide a scientific approach to the development of more usable interactive systems. We outline now the contributions of this thesis toward that goal.

Formality rarely provides its own context; it is often necessary to provide informal motivation for the formalism. This motivation is usually culled from common sense about the world around us. The interaction framework is such a common sense description of interaction in the everyday world. Its description is free of psychological or computational jargon. The framework has served two major purposes in our own work. It has provided the context for our research, enabling us to compare and classify previous HCI research as well as our own. It has also motivated the development of the agent formalisms used in this thesis to describe relevant interactive properties.

We consider interactive systems as a collection of communicating stimulus-response agents. A simple model of an agent was described which was consistent with the interaction framework and allowed for the precise description of several interactive properties. These properties were motivated by an informal description of the translations made between the major components in an interactive system—the *User*, *System*, *Input* and *Output*. We provided an account of how those translations affect the overall goal-directed behaviour of the human user. Some formal descriptions were aimed at describing the ease and coverage of translations in the framework which are themselves seen as implicit agent descriptions based on their stimulus-response behaviour. The correspondence property was described as a relationship between two separate agents that would eventually be embodied by a third translation agent. Other more abstract properties, such as predictability, synthesis and consistency, classified agents by the relationship between their input history and their state or response history.

The problem with the simple agent model was that its near black-box description did not allow for a constructive discussion of interactive properties that relate the goals or results of the interaction with the immediate and visible information that the user sees. At the abstract level of the simple agent, these properties can only be discussed by use of mappings from the state space to the respective result and display spaces, as is done in the red-PIE model and Sufrin and He's model

of interactive processes. In designiug a system, however, we need to be able to construct the agents from the result and display information of a task analysis.

So motivated by the desire to express interactive properties more constructively, we refiued the agent model. The refinement was intended to support a hybrid view of an agent, using complementary descriptious of its internal, state-based behaviour and its external, event-based behaviour. The state of the agent was given as an attribute-value mapping and the two complementary views were made consistent by a communication description which linked the operations of the internal description with the events of the external description that occur on input, output, synchronized and internal chaunels. We also defined two composition operators, corresponding to the synchronous combination of independent agents (those with no attributes in common) and the interleaved combination of dependent agents (those with attributes in common).

The description of the refined ageut model was produced in the Z notation, but we felt that Z did not provide a natural means for building the description of an agent, i.e., Z was not a suitable design notation. We, therefore, provided a language for ageuts which made clear the internal, external and communication descriptions. Oue criterion for developing this language was that it be familiar to those with experience in other formal notations, and so it was made to look similar to a model-oriented uotation, such as Z or VDM, for the internal description, and similar to a process algebra notation, such as CSP or CCS, for the external description.

The refined agent model aud its associated language were then used to show how interactive properties relating result and display information could replace the previous more abstract properties on agents. Templates have been previously used in abstract models as a means of modelling the focus of attention of a user. We gave a coustructive defiuition of templates for restricting the state of an agent to those attributes which are relevant to a particular task. The identification of result and display templates as task-dependeut descriptious derived from task analysis links agent design in a more user-centred interactive system development method. Reformulations of interactive properties related input to task-specific result and display templates yielded versions of predictability, consistency, etc., that were more relevant to the user's understanding of the interaction.

The final contribution of this thesis was to initiate the formal description of multiagent architectures which have been previously used as heuristic guidelines for interactive system development. Architectures are realistic platforms for the description of implementations, and it is necessary to show how the properties derived in the abstract can be traced down to this more concrete level. We discussed two known multiagent architectures, MVC and PAC, and highlighted a difference between them. PAC provides for the explicit description of correspondence between the *System* (model or abstraction) and *Interface* (view or presentation) agents, whereas MVC does not. We then showed how this feature of the control agent in

PAC is a manifestation of the abstract correspondence property expressed between agents. A more extensive case study then showed how the agent description of the *System* leads to a natural agent description of the *Interface*, which was then used to analyze the actual graphical interfaces to existing interactive systems to explain confirmed user confusion or irritation.

9.3 Future work

Though the contributions of this thesis as stated above are significant, another real contribution of this work is the research agenda that it motivates. We will summarize the main aims of this agenda and then comment on improvements to the current work which are necessary based on our experience using agents.

The agent model is intended as the formal system modelling notation for the ESPRIT Basic Research Action project 3066 (AMODEUS) [13]. Within the project, system modelling fits into a larger research scheme with very courageous aims. The AMODEUS project is interested in assimilating different HCI modelling techniques in order to see how they can be coordinated toward a more effective design practice. The modellers fall into three main categories. There are psychologists whose main research is in the development of models of the user. There are computer scientists whose main interest in in the development of models for system description. These two modelling domains are linked by a third domain concerned with how practicing interactive systems developers document the rationale behind their design decisions.

As briefly discussed in Chapters 7 and 8, the main exercises within the project are based on separate analyses of scenarios of interaction between user and system. These scenarios are used as a means of eliciting information comparing and contrasting the utility of the different modelling domains. Earlier versions of the agent model and notation have been used with relative success by the system modellers, and some of those examples have been provided within this thesis. Many more scenarios have been investigated [72, 8, 42], and further are planned. Some of the case studies address issues in computer-supported cooperative work (CSCW) and highly interactive display-based systems. Continued application of the agent model to these scenarios in case studies specifically geared to test its ability to capture relevant information concerning interactive behaviour will undoubtedly lead to further refinements of the model and an increased confidence in its utility.

We realize that the formal agent model cannot alone address all of the important aspects of interaction, not even from the restricted viewpoint of system modelling. For example, the agent model does not provide a way of determining the relevant agents in the system from some requirements specification or scenario description. Therefore, a number of system modellers are working on ways of applying design heuristics in the form of an expert system design tool to help a designer determine the agents and their communications connections before embarking on a formal

description [70]. We also plan to investigate the possibility of applying the agent model more within the user modelling domain as a way of expressing the results of psychological theory in a language of design.

Refinement from specification toward implementation within the agent model should also be possible. Sufrin and He demonstrated how a refinement ordering can be defined on their model of interactive processes, leading to a definition of operational and data refinement in terms of the result and display behaviour. With slight modifications, this definition of the refinement ordering can be mimicked in the agent model, with the additional benefit that refinement can be restricted to task, since the result and display information in the agent model is defined in terms of task. The agent language could be augmented to wide spectrum language so that refinement can remain within the agent model and proceed from high-level specification to implementation, as described by Morgan's refinement calculus on Dijkstra's guarded command language [116, 115].

We mentioned in Chapter 5 that the inclusion of the simplest traces model of a process algebra was intended as an example of how more complicated process algebras could be incorporated. We have not investigated this point very thoroughly, and it would be valuable to see what advantages actually arise from the use of other process algebra models. We suspect that the inclusion of more sophisticated models, such as the failures-divergence model of CSP, would not automatically introduce the ability to express liveness properties because of the constraint within the model that both internal and external descriptions determine overall behaviour. Whereas the external description of an agent in a failures-divergence semantics may be shown to satisfy some liveness properties—guaranteeing that some desirable behaviour will happen—the corresponding internal description may prohibit the agent from engaging in the desired property.

An interesting comment on the development of the agent model is that it was directly influenced by two different perspectives on its behaviour, the internal and the external. The need for a hybrid notation arose when it was realized that the two different perspectives corresponded to different natural notations. Hence, a model-oriented language more naturally expressed internal behaviour and a process algebra notation more naturally expressed external behaviour. Research at York is now investigating a classification of perspectives on system models in order to determine how hybrid notations can be adapted to express the different perspectives [16].

We first described the behaviour of an agent as stimulus-response. In moving toward a concrete notation that corresponded to both Z and CSP, we lost the ability to simply express the connection between a state transition, the stimuli that triggered it and the responses that resulted from it. We admit that this connection is much more naturally expressed within standard Z by the use of decorations ? and ! to mark input and output respectively. However, the primitive communication

operators in Z do not allow the arbitrary communication between the output of one operation and the input to other operations, and so we abandoned the use of Z exclusively. We are not familiar with any attempts in the Z community which have solved this communication problem. However, it may be possible to define in Object-Z a communication component to the object class, allowing for a calculus of objects. We hope to pursue this point by using the agent model as an example of how communication can be incorporated into a model-oriented notation.

In summary, we see the work of this thesis as a solid justification for the use of formal methods in system development to promote non-functional as well as functional requirements. Detailed examination of the non-functional requirements related to usability have shown that they can be addressed scientifically to answer relevant research questions. In addition, we point out that the application of formal methods to areas such as HCI can indicate how to improve the existing formal notations themselves.

Bibliography

[1] G. Abowd, A. Dix, and M. Harrison. State of the art: Formal aspects of user interfaces. Internal report, Human-Computer Interaction Group, Department of Computer Science, University of York. Presented at Eurographics'90. May 1990.

[2] G. Abowd, A. Dix, and M. Harrison. Formalising user recognisable structures of graphics packages. In *Proceedings of the Eurographics Workshop on Formal Methods in Computer Graphics*, Marina di Carrara, Italy, June 1991.

[3] G. D. Abowd. Properties of a graphical interface within a formal interactive system architecture. In *Proceedings of the Eurographics Workshop on Formal Methods in Computer Graphics*, Marina di Carrara, Italy, June 1991.

[4] G. D. Abowd and R. Beale. A framework for the analysis and design of interactive systems. Technical Report YCS (156), University of York, Department of Computer Science, 1991.

[5] G. D. Abowd and R. Beale. Users, systems and interfaces: a unifying framework for interaction. In D. Diaper and N. Hammond, editors, *HCI'91: Usability Now*. British Computer Society Special Interest Group on Human-Computer Interaction, Cambridge, 1991.

[6] G. D. Abowd, J. Bowen, A. Dix, M. Harrison, and R. Took. User interface lauguages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory Programming Research Group, October 1989. Also published as internal report 2487-TN-PRG-1008 Issue 1.0 for ESPRIT project 2487 (REDO).

[7] G. D. Abowd and M. D. Harrison. On a constructive approach to applying formal methods in HCI. Technical Report YCS (151), University of York, Department of Computer Science, December 1990.

[8] G. D. Abowd and M. D. Harrison. Design scenarios for M1.5. Working paper RP1/WP10, Esprit BRA project 3066 (AMODEUS), March 1991.

[9] G. D. Abowd, M. D. Harrison, and C. R. Roast. Modelling predictability in interactive systems. Working paper RP1/WP9, Esprit BRA project 3066 (AMODEUS), September 1990.

[10] H. Alexander. Executable specifications as an aid to dialogue design. In H. J. Bullinger and B. Shackel, editors, *Human-Computer Interaction — INTERACT'87*, pages 739–744. North Holland, 1987.

[11] H. Alexander. Formally-based techniques for designing human-computer dialogues. In D. Diaper and R. Winder, editors, *People and Computers III*, pages 201–214. Cambridge University Press, 1987.

[12] H. Alexander. *Formally-Based Tools and Techniques for Human-Computer Dialogues.* Ellis Horwood Ltd., 1987.

[13] AMODEUS consortium. Assimilating models of designers, users and systems. Esprit Basic Research Action 3066, Technical Annex, 1989.

[14] S. O. Anderson. Proving properties of interactive systems. Technical report, Department of Computer Science, Heriot-Watt University, 1985.

[15] S. O. Anderson. Proving properties of interactive systems. In M. D. Harrison and A. F. Monk, editors, *People and Computers: Designing for Usability.* Cambridge University Press, 1986.

[16] P. Andrews. First year report on doctoral research. Department of Computer Science, University of York, June 1991.

[17] P. Barnard. Interacting cognitive subsystems: A psycholinguistic approach to short-term memory. In A. Ellis, editor, *Progress in the psychology of language*, volume 2, chapter 6. Lawrence Erlbaum, 1985.

[18] P. Barnard. Cognitive resources and the learning of human-computer dialogs. In J. M. Carroll, editor, *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, pages 112–158. MIT Press, 1987.

[19] P. Barnard and M. Harrison. Towards a framework of modelling interactions. Project deliverable D3, Esprit BRA project 3066 (AMODEUS), September 1990.

[20] P. Barnard and M. D. Harrison. Integrating cognitive and system models in human computer interaction. In A. G. Sutcliffe and L. A. Macauley, editors, *People and Computers V.* Cambridge University Press, 1989.

[21] P. Barnard, M. Wilson, and A. MacLean. Approximate modelling of cognitive activity with an expert system: A theory-based strategy for developing an interactive design tool. *The Computer Journal*, 31(5):445–456, 1988.

[22] P. J. Barnard and M. D. Harrison. Towards a framework for modelling human-computer interaction. Working paper RP3/WP5, Esprit BRA project 3066 (AMODEUS), May 1991. Submitted to Esprit Confereuce 1991.

[23] L. Bass, R. Little, R. Pellegrino, S. Reed, R. Seacord, S. Sheppard, and M. R. Szczur. The arch model: Seeheim revisited. User Interface Developer's Workshop Report, April 1991.

[24] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International, 1988.

[25] R. Bornat. Imperative languages in distributed computing. In D. A. Duce, editor, *Distributed Computing Systems Programme*, IEE Digital Electronics and Compnting Services, 1984.

[26] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. Internal report from Tunis University. 1990.

[27] J. P. Bowen, R. B. Gimson, and S. Topp-Jørgensen. Specifying system implementations in Z. Technical Monograph PRG-63, Oxford University Computing Laboratory Programming Research Group, February 1988.

[28] S. D. Brookes. *A Model for Communicating Sequential Processes.* D.Phil. thesis, Oxford University, 1983. Also published as Carnegie-Mellon Technical Report CMU–CS–83–149.

[29] S. Burbeck. How to use model-view-controller (MVC). Techuical report, Softsmarts, Inc., 1987.

[30] K. Butler, J. Bennett, P. Polson, and J. Karat. Report on the workshop on analytical models: Predicting the complexity of human-computer interaction. *SIGCHI Bulletin*, 20(4):63–79, April 1989.

[31] S. K. Card, J. D. Mackinlay, and G. G. Robertsou. The design space of input devices. In J. C. Chew and J. Whiteside, editors, *Empowering People, Proceedings of CHI'90 Conference*, pages 117–124, 1990.

[32] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human Computer Interaction.* Lawrence Erlbaum, 1983.

[33] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In S. P. Robertson, G. M. Olson, and J. S. Olsou, editors, *Reaching Through Technology, Proceedings of the CHI'91 Conference.* pages 181–188. ACM Press, 1991.

[34] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Formal specification in Object-Z: Introduction and case studies. Technical Report 105, Key Centre for Software Technology, Dept. of Computer Science, University of Queensland, July 1989.

[35] J. M. Carroll. Infinite detail and emulation in an ontologically minimized HCI. In J. C. Chew and J. Whiteside, editors, *Empowering People, Proceedings of CHI'90 Conference*, pages 321–327, 1990.

[36] J. M. Carroll and M. B. Rosson. Usability specifications as a tool in iterative development. In R. Hartson, editor, *Advances in Human-Computer Interaction* Ablex, 1984.

[37] J. Coenen, W.-P. de Roever, and J. Zwiers. Assertional data reification proofs: Survey and perspective. In *Fourth Refinement Workshop of the BCS FACS Special Interest Group*, January 1991.

[38] J. Coutaz. PAC, an object oriented model for dialog design. In H. J. Bullinger and B. Shackel, editors. *Human-Computer Interaction — INTERACT'87.* pages 431–436. North-Holland, Amsterdam, 1987.

[39] J. Coutaz. *Interface Homme-Ordinateur: Conception et Réalisation.* PhD thesis, University of Grenoble, Laboratoire de Génie Informatique, December 1988.

[40] J. Coutaz. Architecture models for interactive software. In *Proc. of ECOOP'89*, 1989.

[41] J. Coutaz. Architecture models for interactive software: Failures and trends. In *Engineering for Human-Computer Interaction*, 1989. Proceedings of the IFIP WG 2.7 conference.

[42] J. Coutaz, L. Nigay, M. Harrison, and G. Abowd. Design scenarios for m1.5. Working paper RP2/WP11, Esprit BRA project 3066 (AMODEUS), March 1991.

[43] B. J. Cox. *Object-Oriented Programming: an Evolutionary Approach.* Addison-Wesley, 1986.

[44] M. Curry, A. Monk, and P. Wright. Obstacles to the use of formal notations in software design practice. In University of York, Department of Computer Science Technical Report, September 1990.

[45] J. Davies. *Specification and Proof: Real-Time Systems.* D.Phil. thesis, Oxford University, 1991.

[46] J. Davies and S. Schneider. An introduction to timed CSP. Technical Monograph PRG-75, Oxford University Computing Laboratory Programming Research Group, August 1989.

[47] A. Dix. Nondeterminism as a paradigm for understanding the user interface. In M. D. Harrison and H. W. Thimbleby, editors, *Formal methods in Human-Computer Interaction*, Cambridge Series on Human-Computer Interaction, chapter 4. Cambridge University Press, 1990.

[48] A. J. Dix. *Formal Methods and Interactive Systems: Principles and Practice.* D.Phil. thesis, University of York, 1987.

[49] A. J. Dix. *Formal Methods for Interactive Systems.* Academic Press, 1991.

[50] A. J. Dix and M. D. Harrison. Principles and interaction models for window managers. In M. D. Harrison and A. F. Monk, editors, *People and Computers: Designing for usability*, pages 352–366. Cambridge University Press, 1986.

[51] A. J. Dix and M. D. Harrison. Interactive systems design and formal development are incompatible? In J. A. McDermid, editor, *Proceedings 1988 Refinement Workshop.* Butterworth Scientific, 1989.

[52] A. J. Dix, M. D. Harrison, and E. E. Miranda. Using principles to design features of a small programming environment. In I. Sommerville. editor, *Software Engineering Environments*, pages 135–150. Peter Peregrinus, 1986.

[53] A. J. Dix, M. D. Harrison, C. Runciman, and H. W. Thimbleby. Interaction models and the principled design of interactive systems. In H. Nichols and D. S. Simpson, editors, *European Software Engineering Conference*, pages 127–135. Springer Lecture Notes, 1987.

[54] A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *People and Computers: Designing the interface*, pages 13–22. Cambridge University Press, 1985.

[55] D. Duke and R. Duke. A history model for classes in Object-Z. Technical Report 120, Key Centre for Software Technology, Dept. of Computer Science, University of Qneensland, November 1989.

[56] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Key Centre for Software Technology, Dept. of Computer Science, University of Queensland, May 1991.

[57] J. Finlay, A. Green, P. Barnard, and M. Harrison. Linking user and system models: an interaction structure. Working paper RP3/WP2, Esprit BRA project 3066 (AMODEUS), January 1991.

[58] FOCUS. Fonndations of opto-electronic computer systems, action 3180. In *Synopses of Basic Research: actions, working groups and networks of excellence, Volume 8*, pages 192–195, January 1991.

[59] J. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics.* Addison Wesley, 1984.

[60] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness aud implementation of abstract data types. In Yeh, editor, *Current Trends in Programming Methodology*, volume IV: Data Strncturing, pages 80–149. Prentice-Hall, 1978.

[61] A. Goldberg. *Smalltalk-80: The Interactive Programming Enviornment.* Addison-Wesley, 1983.

[62] M. Green. The University of Alberta user interface management system. *ACM Conference of the Speeial Interest Group for Graphics*, 19(3):205–214, July 1985.

[63] T. R. G. Green, F. Schiele, and S. J. Payne. Formalisable medels of nser knowledge in human-computer interaction. In G. C. vad der Veer, T. R. G. Green, J.-M. Hoc, and D. Murray, editors, *Working with Computers: Theory versus Outcome*, Compnters and People Series, chapter 1. Academic Press, 1988.

[64] J. Grudin. The case against user interface consistency. *Communications of the ACM*, 4(3):245–264, 1989.

[65] J. Guttag and J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[66] M. Hanlon and J. Newton. FOREST case study: The MVC. Technical report, Advanced Techniques Group, Data Logic Limited, Queens House, Greenhill Way, Harrow, October 1990. 1990.

[67] D. Harel. Statecharts: a visnal formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[68] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514-530, May 1988.

[69] M. Harrison and G. Abowd. Formal methods in human-computer interaction: a tutorial. Technical Report YCS 155, University of York, Department of Computer Science, 1991. Also a tutorial presentation at CHI'91 conference.

[70] M. Harrison, J. Coutaz, L. Nigay, J. E. Finlay, and G. D. Abowd. Agent architectures and the application of transformation rules in interactive system development. Project deliverable D2, Esprit BRA project 3066 (AMODEUS), September 1990.

[71] M. D. Harrison. Modelling user structures within system specifications. In Colloquium on Formal Methods in HCI: III, sponsored by IEE Professional Group C5 (Man-Machine Interaction). December 1989.

[72] M. D. Harrison, J. Coutaz, J. E. Finlay, G. D. Abowd, and L. Nigay. Interaction analysis techniques from a system modelling viewpoint. Project deliverable D1, Esprit BRA project 3066 (AMODEUS), September 1990.

[73] M. D. Harrison and A. J. Dix. A state model of direct manipulation. In M. D. Harrison and H. W. Thimbleby, editors, *Formal Methods in Human Computer Interaction*, pages 129–151. Cambridge University Press, 1990.

[74] M. D. Harrison, C. R. Roast, and P. C. Wright. Complementary methods for the iterative design of interactive systems. In G. Salvendy and M Smith, editors, *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, pages 651–658. Elsevier Scientific, 1989.

[75] M. D. Harrison and H. W. Thimbleby, editors. *Formal Methods in Human Computer Interaction*. Cambridge University Press, 1990.

[76] J. He. Process refinement. In J. A. McDermid, editor, *The Theory and Practice of Refinement*. Butterworth Scientific, 1989. Proceedings of 1988 York Refinement Workshop.

[77] J. He. A trace-state based approach to process specification and design. Oxford University Computing Laboratory Programming Research Group, 1989.

[78] J. He, C. Hoare, and J. Sanders. Data refinement refined. In *European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[79] S. Hekmatpour and D. Ince. Evolutionary prototyping and the human-computer interface. In H.-J. Bullinger and B. Shackel, editors, *Proceedings of INTERACT'87*, pages 479–484. North-Holland, 1987.

[80] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[81] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, 1978.

[82] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[83] A. Howes and S. J. Payne. Display-based competence: towards user models for menu-driven interfaces. University of Lancaster.

[84] J. Hughes. Specifying a visual file system in Z. In *Formal Methods in HCI: III.* IEE Professional Group C5 (Man-machine interaction), 1989.

[85] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. *Human-Computer Interaction*, 1:331–338, 1985.

[86] INMOS Limited. *occam Programming Manual.* Prentice-Hall International, London, 1984.

[87] ISO 8807. Information processing systems—open systems interconnection — LOTOS—a formal technique based on the temporal ordering of observational behaviour. Technical report, ISO Standards Authority, 1988.

[88] R. J. K. Jacob. Survey and examples of specification techniques for user-computer interfaces. Technical report, Naval Research Laboratory, Washington, D.C., 1983.

[89] R. J. K. Jacob. Using formal specifications in the design of a human-computer interface. *Communications of the ACM*, 26(4):259–264, 1983.

[90] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[91] G. Jones. *Programming in occam.* Prentice-Hall International, 1986.

[92] M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3, 1988.

[93] D. E. Kieras and P. G. Polson. An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*, 22:365–394, 1985.

[94] C. Knowles. Can cognitive complexity theory (CCT) produce an adequate measure of system usability? In D. M. Jones and R. Winder, editors, *People and Computers IV*, pages 291–307. Cambridge University Press, 1988.

[95] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 23, 1983.

[96] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *JOOP*, 1(3), August 1988.

[97] S. Krening. The structure of interaction. Working paper RP3/WP6, Esprit BRA project 3066 (AMODEUS), June 1991.

[98] J. Laird, A. Newell, and P. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[99] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[100] K. Lano. Z^{++}, an object-orientated extension to Z. In *Proceedings of the fifth annual Z User Meeting*, December 1990.

[101] C. H. Lewis. A research agenda for the nineties in human-computer interaction. *Human-Computer Interaction*, 5(2–3):125–143, 1990.

[102] B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–59, April 1974.

[103] R. Macdonald. Z usage and abusage. Report 910003, Royal Signals and Radar Establishment, Malvern, Worcestershire, February 1991.

[104] Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.

[105] L. S. Marshall. *A Formal Description Method for User Interfaces*. PhD thesis, University of Manchester, United Kingdom, 1986. Also published as technical report UMCS-87-1-2.

[106] M. Martins. Formal specification of highly interactive text editors. Technical Report RAL–87–093, Rutherford Appletou Laboratory, November 1987.

[107] A. Mével and T. Guéguen. *Smalltalk-80*. Macmillan Education, 1987.

[108] F. Mili and A. Mili. Relational heuristics for programming: Advances and perspectives. Internal report from Oakland University and Tunis, November 1989.

[109] A. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[110] R. Milner. *Communication and Concurrency*. Prentice-Hall International, UK, Ltd., London, 1989.

[111] F. Moller and C. Tofts. A temporal calculus of communicating systems. LFCS Report Series ECS–LFCS–89–104, LFCS, Department of Computer Science, University of Edinburgh, December 1989.

[112] A. Monk, editor. *Fundamentals of Human-Computer Interaction.* Computers and People Series. Academic Press, 1984.

[113] A. F. Monk and A. J. Dix. Refining early design decisions with a black-box model. In D. Diaper and R. Winder, editors, *People and Compnters III, HCI'87*, pages 147–158. Cambridge University Press, 1987.

[114] T. P. Moran. The command language grammar: a representation for the user interface of interactive systems. *International Journal of Man Machine Systems*, 15, 1981.

[115] C. Morgan. *Programming from Specifications.* Prentice-Hall International, 1990.

[116] C. Morgan, K. Robinson, and P. Gardiner. On the refinement calculus. Technical Monograph PRG-70, Oxford University Computing Laboratory Programming Research Group, October 1988.

[117] C. C. Morgan and B. A. Sufrin. Specification of the Unix filing system. *IEEE Transactions on Software Engineering*, 10(2):128–142, March 1984.

[118] K. Myers and N. Hammond. Defiuition of scenarios for M1 workshop. Internal report IR6, Esprit BRA project 3066 (AMODEUS), July 1990.

[119] A. Newell and H. Simon. *Human Problem Solving.* Prentice-Hall, 1972.

[120] W. Newman. A system for interactive graphical programming. In *SJCC 1968*, pages 47–54, Washington D.C., 1968. Thompson Books.

[121] L. Nigay. Modelisation des architectures logicielles des systems interactifs. Master's thesis, Institut National Polytechnique de Grenoble, Lahoratoire de Génie informatique, June 1990.

[122] D. Norman. Some observations on mental models. In D. Gentner and A. Stevens, editors, *Mental Models*, pages 7–14. Erlbaum, 1983.

[123] D. A. Norman. Four stages of user activities. In B. Shackel, editor, *Human-Computer Interaction—INTERACT'84*, pages 507–511. Elsevier Science Publishers, 1984.

[124] D. A. Norman. Cognitive engineering. In D. A. Norman and S. Draper, editors, *User-Centered System Design*, pages 31–62. Erlbaum, 1986.

[125] D. A. Norman. *The Psychology of Everyday Things*. Basic Books, 1988.

[126] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–, May 1972.

[127] S. J. Payne. Task-action grammars. In B. Shackel, editor, *Human-Computer Interaction—INTERACT'84*, pages 527–532. Elsevier Science Publishers, 1984.

[128] S. J. Payne and T. R. G. Green. Task-action grammars: a model of mental representation of task languages. *Human-Computer Interaction*, 2(2):93–133, 1986.

[129] G. Pfaff and P. ten Hagen, editors. *Seeheim Workshop on User Interface Management Systems*, Berlin, 1985. Springer-Verlag.

[130] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[131] V. Pratt. A decidable mu-calculus. In *Proceedings of the 22nd IEEE Conference on the Foundations of Computer Science*, 1981.

[132] G. Reed. *A Uniform Mathematical Theory for Real-Time Distributed Computing*. D.Phil. thesis, Oxford University, 1988.

[133] P. Reisner. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering*, SE-7(2):229–240, 1981.

[134] P. Reisner. Analytic tools for human factors of software. Research Report RJ 3808 (43605), IBM Research Laboratory, San Jose, 1983.

[135] P. Reisner. Formal grammar as a tool for analysing ease of use: some fundamental concepts. In J. C. Thomas and M. L. Schneider, editors, *Human Factors in Computer Systems*, pages 53–78. Ablex, 1983.

[136] P. Reisner. What is inconsistency. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Human-Computer Interaction—INTERACT'90*, pages 175–181. Elsevier Science Publishers, 1990.

[137] C. R. Roast and P. C. Wright. Incorporating the user's perspective into a system model. Technical Report YCS 148, University of York, Department of Computer Science, 1990.

[138] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In S. P. Robertson, G. M. Olson, and J. S. Olson, editors, *Reaching Through Technology, Proceedings of the CHI'91 Conference*, pages 189–194. ACM Press, 1991.

[139] A. Roscoe. *Denotational Semantics for Occam*, volume 197 of *Lecture Notes in Computer Science*, pages 306–329. Springer-Verlag, 1985. July 1984 Seminar on Concurrency.

[140] A. Roscoe and C. Hoare. The laws of occam programming. Technical Monograph PRG-53, Oxford University Computing Laboratory Programming Research Group, February 1986.

[141] M. Ryan. Developments of MAL in forest research. WP1 (Logic and Language) Deliverable NFR/WP1.1/IC/R/001/A, Forest Research Project, October 1990.

[142] F. Schiele and T. Green. HCI formalisms and cognitie psychology: the case of task-action grammars. In M. D. Harrison and H. W. Thimbleby, editors, *Formal methods in Human-Computer Interaction*, Cambridge Series on Human-Computer Interaction, chapter 2. Cambridge University Press, 1990.

[143] R. L. Schnapp. *SuperPaint 1.1*. Silicon Beach Software, 1988. User's manual for Macintosh software package.

[144] S. A. Schneider. *Correctness and Communication in Real-Time Systems*. D.Phil. thesis, Oxford University, 1989.

[145] S. Schuman, D. Pitt, and P. Byers. Object-oriented process specification. Technical report, Department of Mathematics, University of Surrey, 1990.

[146] S. A. Schuman and D. Pitt. Object-oriented subsystem specification. In L. Meertens, editor, *Program Specification and Transformation*. Elsevier Science Publishers, 1987.

[147] B. D. Sharratt. The incorporation of early interface evaluation into command language grammar specifications. In D. Diaper and R. Winder, editors, *People and Computers III—Proceedings of HCI'87*. Cambridge University Press, 1987.

[148] B. D. Sharratt. Top-down interactive systems design: some lessons learnt from using command language grammar specifications. In H.-J. Bullinger and B. Shackel, editors, *Proceedings of INTERACT'87*, pages 395–399. North-Holland, 1987.

[149] J. L. Sibert, R. Belliardi, and A. Kamran. Some thoughts on the interface between UIMS and application programs. In *User Interface Management Systems*, pages 183–189. Springer-Verlag, 1985.

[150] J. L. Sibert, W. D. Hurley, and T. W. Bleser. An object-oriented user interface management system. In *ACM SIGGRAPH'86*, 1986.

[151] T. Simon. Analysing the scope of cognitive models in human-computer interaction: a trade-off approach. In D. M. Jones and R. Winder, editors, *People and Computers IV*, pages 79–93. Cambridge University Press, 1988.

[152] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 1988.

[153] J. M. Spivey. *Understanding Z, a Specification Language and its Semantics.* Cambridge University Press, 1989.

[154] C. Stanley-Smith and T. Cahill. UNIFORM: A language geared to system independence. Project Document UL-TN-1002, Esprit project REDO. 1989.

[155] C. Stirling and D. Walker. A general tableau technique for verifying temporal properties of concurrent programs. In *Semantics for Concurrency*, Workshops in Computing. Springer-Verlag, 1990. Extended abstract.

[156] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[157] L. Suchman. *Plans and Situated Actions: The Problem of Human Machine Interaction.* Cambridge University Press, 1987.

[158] B. Sufrin and J. He. Specification, refinement and analysis of interactive processes. In M. D. Harrison and H. W. Thimbleby, editors, *Formal methods in Human-Computer Interaction*, Cambridge Series on Human-Computer Interaction, chapter 6. Cambridge University Press, 1990.

[159] A. Sutcliffe. Some experiences in integrating specification of human-computer interaction within a structured system development method. In D. Jones and R. Winder, editors, *People and Computers IV*, pages 145–160. Cambridge University Press, 1988.

[160] H. Thimbleby. Generative user-engineering principles for user interface design. In B. Shackel, editor, *Human-Computer Interaction—INTERACT'84*, pages 661–666. Elsevier Science Publishers, 1984.

[161] H. Thimbleby. *User Interface Design.* Adison Wesley, 1990.

[162] R. K. Took. Surface interaction: A paradigm model for separating application and interface. In *Proceedings of CHI '90*, 1990.

[163] R. K. Took. *Surface Interaction: An Architecture and Formal Model for Separating Application and Interface.* D.Phil. thesis, University of York, 1990.

[164] P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS.* North-Holland, Amsterdam, 1989.

[165] P. Walsh, K. Lim, J. Long, and M. Carver. JSD and the design of user interface software. *Ergonomics*, 1989.

[166] J. Whiteside and D. Wixon. Discussion: Improving human-computer interaction—a quest for cognitive science. In J. Carroll, editor, *Interfacing thought: cognitive aspects of human-computer interaction.* MIT press, 1987.

[167] P. J. Whysall and J. A. McDermid. An approach to object oriented specification using Z. In *Proceedings of the fifth annual Z User Meeting*, December 1990.

[168] P. J. Whysall and J. A. McDermid. Object oriented specification and refinement. In *Fourth Refinement Workshop of the BCS FACS Special Interest Group*, January 1991.

[169] T. Winograd and F. Flores. *Understanding Computers and Cognition.* Addison-Wesley, 1987.

[170] J. Woodcock and C. Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z—Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Seience.* Springer-Verlag, 1990.

[171] J. C. Woodcock. Mathematics as a management tool: Proof rules for promotion. In *Proc. 6th Annual CSR Conference on Large Software Systems*, September 1989.

[172] P. C. Wright and C. R. Roast. Abstraction and generalisation in the analysis of usability. In University of York, Department of Computer Science Technical Report, September 1990.

[173] R. Young and J. Whittington. Using a knowledge analysis to predict conceptual errors in text-editor usage. In J. Chew and J. Whiteside, editors, *CHI'90 Conference Proceedings*, pages 91–97. Addison Wesley, 1990.

[174] R. M. Young and P. Barnard. The use of scenarios in human-computer interaction research: turbocharging the tortoise of cumulative science. In J. Carroll and P. Tanner, editors, *CHI and GI Conference Proceedings: Human Factors in Computing Systems and Graphic Interface*, pages 291–296. ACM, 1987.

[175] R. M. Young, P. Barnard, T. Simon, and J. Whittington. How would your favourite user model cope with these scenarios? *SIGCHI Bulletin*, 20(4):51–55, 1989.

Appendix A

Use of the Z Notation

We make extensive use of the Z notation in this thesis. For the most part, we have adhered to the standard Z notation, as given by Spivey's standard Z reference manual [152]. In this appendix, we discuss some stylistic conventions that we have adopted, and we provide definitions of operators used in the body of the thesis but not defined there nor in the standard Z reference manual.

Function definitions

When defining a function by predicates describing its effect on arguments in its domain, it is technically necessary that these axioms be bound by some universal quantifier over the domain elements. For example, in the Z reference manual, the projection functions on ordered pairs, are defined as below.

$$
\begin{array}{|l}
\hline
[X, Y] \\
\mathit{first} : X \times Y \rightarrow X \\
\mathit{second} : X \times Y \rightarrow Y \\
\hline
\forall x : X;\ y : Y \\
\quad \bullet\ \mathit{first}(x, y) = x \\
\qquad \mathit{second}(x, y) = y \\
\hline
\end{array}
$$

We have decided that in most cases the type information of the function is enough to allow the elimination of the universal quantification, leading to a slightly less cluttered presentation that is no less understandable. The projection functions would be rewritten as below.

$$[X, Y]$$
$$first : X \times Y \rightarrow X$$
$$second : X \times Y \rightarrow Y$$
$$first(x, y) = x$$
$$second(x, y) = y$$

Formatting quantifications

Many quantifications with bonnd variables—universal, existential, set comprehensions, mu-expressions and lambda expressions—are rather lengthy and it is always a challenge to present with maximal clarity and minimal bracketing. In this thesis, we have tried to present a consistent format to these qnantifications. Where one would normally write such a nniversal qnantification as

$$\forall Decls \mid P \bullet Q$$

we now write

$$\forall Decls$$
$$\mid P$$
$$\bullet Q$$

allowing indentation to replace the need for some bracketing.

As mentioned by Macdonald [103], it is common in Z usage to replace existential qnantification by a **where** clause to increase readability. Hence, the expression

$$\exists Decls$$
$$\bullet Q$$

can be replace by

$$Q$$
where *Decls*

Operations on sequences

We will make use of some operations on sequences. The generic relations $\prec$ ("is a prefix of"), $\succ$ ("is a snffix of") and $\preccurlyeq$ ("is contained in") represent useful operations. prefixes is a function which generates the set of all prefixes of a given

sequence. The prefix relation prefix_closed is true if its argument is a set of sequences which is closed under the prefix relation, that is, the prefixes of every sequence in the set are also in the set.

$$
\begin{array}{|l}
\hline
[X] \\
\hline
\rightarrowtail : \operatorname{seq} X \leftrightarrow \operatorname{seq} X \\
\twoheadrightarrow : \operatorname{seq} X \leftrightarrow \operatorname{seq} X \\
\rightleftharpoons : \operatorname{seq} X \leftrightarrow \operatorname{seq} X \\
\operatorname{prefixes} : \operatorname{seq} X \rightarrow \mathbb{P} \operatorname{seq} X \\
\operatorname{prefix\_closed} \_ : \mathbb{P} \operatorname{seq} X \\
\hline
xs \rightarrowtail ys \Leftrightarrow \exists zs : \operatorname{seq} X \bullet xs \frown zs = ys \\
xs \twoheadrightarrow ys \Leftrightarrow \exists zs : \operatorname{seq} X \bullet zs \frown xs = ys \\
xs \rightleftharpoons ys \Leftrightarrow \exists zs, zs' : \operatorname{seq} X \bullet zs \frown xs \frown zs' = ys \\
\operatorname{prefixes} ys = \{ xs : \operatorname{seq} X \mid xs \rightarrowtail ys \} \\
\operatorname{prefix\_closed} XS \Leftrightarrow \forall xs' : XS \bullet \operatorname{prefixes} xs' \subseteq XS \\
\hline
\end{array}
$$

Technically, the definition of these operations as generic mandates that their use be indexed by the base set, so we would have to say $xs \rightarrowtail [Y] ys$ to indicate that xs is a prefix of ys, where both are sequences from the set Y. In practice, we will not indicate the base set when it is clear from the context.

⨾/ (Distributed sequential composition)

The function ⨾/ sequentially composes a sequence of homogeneous relations (relations of the form $X \leftrightarrow X$) to obtain one relation.

$$
\begin{array}{|l}
\hline
[X] \\
\hline
⨾/ : \operatorname{seq}(X \leftrightarrow X) \rightarrow (X \leftrightarrow X) \\
\hline
⨾/\langle\,\rangle = \operatorname{id} X \\
⨾/\langle R \rangle = R \\
⨾/(\langle R \rangle \frown rs) = R \fatsemi (⨾/ rs) \\
\hline
\end{array}
$$

Sequence filtering

As described in Sufrin and He [158], we can extend the notion of domain and range restriction to act on sequences. These operators will behave like the familiar filtering operations in functional programming and used for trace semantics. The expression $N \overset{\text{seq}}{\lhd} s$ will give a sequence consisting of the resequencing of the restriction of the domain of s to the set of natural numbers in N. Similarly, the expression $s \overset{\text{seq}}{\rhd} S$

will give a sequence consisting of the resequencing of the restriction of the range of s to the set of values in S.

$$
\begin{array}{|l}
[X] \\
\hline
resequence : (\mathbb{N} \pfun X) \to \seq X \\
sort : \power \mathbb{N} \to \seq \mathbb{N} \\
increasing : \power(\seq \mathbb{N}) \\
\mathrel{\overset{seq}{\lhd}} : (\power \mathbb{N} \times \seq X) \to \seq X \\
\mathrel{\overset{seq}{\rhd}} : (\seq X \times \power X) \to \seq X \\
\hline
\forall zs : \seq X;\ N : \power \mathbb{N};\ S : \power X \bullet \\
\qquad N \mathrel{\overset{seq}{\lhd}} s = resequence(N \lhd s) \\
\qquad \land\ s \mathrel{\overset{seq}{\rhd}} S = resequence(s \rhd S) \\
\forall f : \mathbb{N} \pfun X \bullet resequence(f) = sort(\dom f) \fatsemi f \\
\forall N : \power \mathbb{N};\ s : \seq X \bullet \\
\qquad \ran sort(N) = N \\
\qquad \land\ sort(N) \in increasing \\
\qquad \land\ s \in increasing \Leftrightarrow \forall i, j : \dom s \bullet i < j \Rightarrow s(i) < s(j)
\end{array}
$$

For example,

$$\{1, 5, 3, 11\} \mathrel{\overset{seq}{\lhd}} \langle a, b, c, d, e, f \rangle = \langle a, c, e \rangle$$

$$\langle a, b, c, d, e, f \rangle \mathrel{\overset{seq}{\rhd}} \{c, g, a\} = \langle a, c \rangle.$$

Some specific filtering on traces as suggested by Hoare [82] will be notationally convenient. We first describe channel filtering, which gives the sequence of messages communicated along a channel. We write $t \downarrow CS$ to represent the restriction of trace t to events on channels in the set CS.

$$
\begin{array}{|l}
\downarrow : (\seq Event \times \power ChannelID) \seq Event \\
\hline
t \downarrow CS = t \fatsemi \mathrm{mess} \mathrel{\overset{seq}{\rhd}} C
\end{array}
$$

Folding

We define general folding operations *foldl* and *foldl*1 with their standard definition from functional programming [24].

$$
\begin{array}{|l}
\hline
[X, Y] \\
\hline
foldl : ((X \times Y) \rightarrow X) \rightarrow X \rightarrow \mathrm{seq}\, Y \rightarrow X \\
foldl1 : ((X \times X) \rightarrow X) \rightarrow \mathrm{seq}\, X \rightarrow X \\
\hline
\forall \_\ op\ \_ : (X \times Y) \rightarrow X;\ a : X;\ y : Y;\ ys : \mathrm{seq}\, Y \\
\quad \bullet\ (\ \ foldl\ op\ a\ \langle\,\rangle = a \\
\qquad \wedge\ foldl\ op\ a\ y \frown ys = foldl\ op\ (a\ op\ y)\ ys) \\
\forall \_\ op\ \_ : (X \times X) \rightarrow X;\ xs : \mathrm{seq}_1\, X \\
\quad \bullet\ foldl1\ op\ xs = foldl\ op\ (head\ xs)\ (tail\ xs) \\
\hline
\end{array}
$$

Similar definitions of the folding operations *foldr* and *foldr*1 can be defined.

Interleaving

Interleaving of sequences is used to define some of the trace semantics for the agent model. We here define interleaving recursively as done by Hoare[82, p. 56].

$$
\begin{array}{|l}
\hline
[X] \\
\hline
interleaves : \mathrm{seq}\, X \leftrightarrow (\mathrm{seq}\, X \times \mathrm{seq}\, X) \\
\hline
\langle\,\rangle\ interleaves\ (t, t') \Leftrightarrow (t = \langle\,\rangle) \wedge (t' = \langle\,\rangle) \\
s\ interleaves\ (t, t') \Leftrightarrow s\ interleaves\ (t', t) \\
(\langle x \rangle \frown s)\ interleaves\ (t, t') \Leftrightarrow \\
\quad (\ \ (t \neq \langle\,\rangle \wedge head(t) = x \wedge s\ interleaves(tail(t), t')) \\
\quad\ \ \vee (t' \neq \langle\,\rangle \wedge head(t') = x \wedge s\ interleaves(t, tail(t')))) \\
\hline
\end{array}
$$

Equivalence relations

An equivalence relation on a type X is relation which is reflexive, symmetric and transitive. We define the generic predicate equivalence[X] to be true when its argument is an equivalence relation on the type X.

$$
\begin{array}{|l}
\hline
[X] \\
\hline
\mathrm{equivalence} : (X \leftrightarrow X) \rightarrow Boolean \\
\hline
\forall \sim : X \leftrightarrow X \bullet \mathrm{equivalence} \sim \ \Leftrightarrow \\
\quad \forall p, q, r : X \bullet\ (\ \ p \sim p \\
\qquad\qquad\qquad \wedge\ \ p \sim q \Rightarrow q \sim p \\
\qquad\qquad\qquad \wedge\ \ (p \sim q \wedge q \sim r) \Rightarrow p \sim r) \\
\hline
\end{array}
$$

As usual, in practice, we will leave out the base type indication when we use equivalence when it is clear from the context.

The set of equivalence classes derived from an equivalence relation $\sim$ on the type X is represented by $classes(\sim)$.

$$
\begin{array}{|l}
\hline
[X] \\
classes : (X \leftrightarrow X) \pfun \mathbb{P}\,\mathbb{P}\,X \\
\hline
\mathrm{dom}\ classes = \{ \sim : X \leftrightarrow X \mid \mathrm{equivalence}\ \sim \} \\
\forall \sim : X \leftrightarrow X \mid \sim \in \mathrm{dom}\ classes \bullet \\
\quad classes(\sim) = \{\, x : X \bullet \{\, x' : X \mid x' \sim x \,\} \,\} \\
\hline
\end{array}
$$

Appendix B

Some theorems on the refined agent model

In this appendix, we provide the complete proofs for some theorems expressed in the body of this thesis. To assist in these proofs, we will need the following result.

Recall the definition of *compatible*, which relates states which agree over their common attributes.

$$\begin{array}{|l} \textit{compatible} : \textit{State} \leftrightarrow \textit{State} \\ \hline (s_1, s_2) \in \textit{compatible} \Leftrightarrow s_1 \oplus s_2 = s_1 \cup s_2 \end{array}$$

The relation *compatible* is reflexive and commutative. A less common but important property of *compatible* is that if two states are compatible, and a third state is compatible with the union of the first two, then the third state is compatible with each of the other states. We state and prove this result as Lemma B.1.

Lemma B.1

$$\begin{array}{l} \forall s, t, u : \textit{State} \\ \quad | \quad (\quad (s, t) \in \textit{compatible} \\ \qquad \wedge (s \cup t, u) \in \textit{compatible}) \\ \quad \bullet \quad (\quad (t, u) \in \textit{compatible} \\ \qquad \wedge (s, u) \in \textit{compatible}) \end{array}$$

PROOF OF LEMMA B.1:

Under the assumptions, we can pick $s, t, u : \textit{State}$ satisfying

$$\begin{array}{ll} (s \cup t) \oplus u = (s \cup t) \cup u & \text{[assumption and defn. of } \textit{compatible}\text{]} \\ \Leftrightarrow (s \oplus t) \oplus u = (s \cup t) \cup u & \text{[}(s, t) \in \textit{compatible}\text{]} \\ \Leftrightarrow s \oplus (t \oplus u) = s \cup (t \cup u) & \text{[accoc. of } \oplus \text{ and } \cup\text{]} \end{array}$$

By case analysis on $A \subseteq \mathcal{A}$, we can satisfy the inference

$$\left(\begin{array}{l} s \oplus t = s \cup t \\ \wedge\, s \oplus (t \oplus u) = s \cup (t \cup u) \end{array} \right) \Rightarrow (t, u) \in \mathit{compatible}$$

Case 1: $A = \mathrm{dom}\, s \cap \mathrm{dom}\, t$

Since $(s, t) \in \mathit{compatible}$, s and t agree on A, so

$$A \lhd (s \oplus (t \oplus u)) = A \lhd (t \oplus u)$$
$$A \lhd (s \cup (t \cup u)) = A \lhd (t \cup u)$$

Therefore, $(t, u) \in \mathit{compatible}$ in this case.

Case 2: $A = \mathrm{dom}\, t - \mathrm{dom}\, s$

In this case, we satisfy

$$A \lhd (s \oplus (t \oplus u)) = A \lhd (t \oplus u)$$
$$A \lhd (s \cup (t \cup u)) = A \lhd (t \cup u,)$$

and so $(t, u) \in \mathit{compatible}$ in this case.

Case 3: $A = \mathrm{dom}\, s - \mathrm{dom}\, t$

In this case, we satisfy

$$A \lhd (t \oplus u) = A \lhd u$$
$$A \lhd (t \cup u) = A \lhd u,$$

and so $(t, u) \in \mathit{compatible}$ in this case.

Case 4: $A = \mathcal{A} - (\mathrm{dom}\, s \cup \mathrm{dom}\, t)$

In this case, we have

$$A \lhd (t \oplus u) = A \lhd u$$
$$A \lhd (t \cup u) = A \lhd u,$$

and so $(t, u) \in \mathit{compatible}$ in this case.

Since these cases are exhaustive, we conclude that $(t, u) \in \mathit{compatible}$. By similar arguments, we can show that $(s, u) \in \mathit{compatible}$.
$\diamond$ END OF PROOF OF LEMMA B.1

Theorem 5.1

$$\begin{array}{l} \forall S1, S2, S3 : \mathit{Stateset} \\ \mid \ (\ \ (S1, S2) \in \mathrm{dom}\, \mathit{join} \\ \quad \wedge (\mathit{join}(S1, S2), S3) \in \mathrm{dom}\, \mathit{join} \\ \bullet \ \mathit{join}(\mathit{join}(S1, S2), S3) = \mathit{join}(S1, \mathit{join}(S2, S3)) \end{array}$$

PROOF OF THEOREM 5.1:

The hypotbesis of this theorem ensures that $\mathit{join}(\mathit{join}(S1, S2), S3)$ exists. The following predicates are true under this hypothesis.

$$\begin{array}{l} \forall a : S1.\mathit{attribs} \cap S2.\mathit{attribs} \\ \quad \bullet \ S1.\mathit{type}(a) = S2.\mathit{type}(a) \\ \exists s_1 : S1.\mathit{states};\ s_2 : S2.\mathit{states} \\ \quad \bullet \ (s_1, s_2) \in \mathit{compatible} \\ \forall a : \mathit{join}(S1, S2).\mathit{attribs} \cap S3.\mathit{attribs} \\ \quad \bullet \ \mathit{join}(S1, S2).\mathit{type}(a) = S3.\mathit{type}(a) \\ \exists s_{12} : \mathit{join}(S1, S2).\mathit{states};\ s_3 : S3.\mathit{states} \\ \quad \bullet \ (s_{12}, s_3) \in \mathit{compatible} \end{array}$$

Under these assumptions, we mnst show that $\mathit{join}(S1, \mathit{join}(S2, S3))$ also exists and is identical. The proof, therefore, is split into two parts.

First, to show that $(S1, \mathit{join}(S2, S3)) \in \mathrm{dom}\, \mathit{join}$, we must satisfy the following four predicates.

$$\begin{array}{ll} \forall a : S2.\mathit{attribs} \cap S3.\mathit{attribs} & \text{(B.1)} \\ \quad \bullet \ S2.\mathit{type}(a) = S3.\mathit{type}(a) & \\ \exists s_2 : S2.\mathit{states};\ s_3 : S3.\mathit{states} & \text{(B.2)} \\ \quad \bullet \ (s_2, s_3) \in \mathit{compatible} & \\ \forall a : S1.\mathit{attribs} \cap \mathit{join}(S2, S3).\mathit{attribs} & \text{(B.3)} \\ \quad \bullet \ S1.\mathit{type}(a) = \mathit{join}(S2, S3).\mathit{type}(a) & \\ \exists s_1 : S1.\mathit{states};\ s_{23} : \mathit{join}(S2, S3).\mathit{states} & \text{(B.4)} \\ \quad \bullet \ (s_1, s_{23}) \in \mathit{compatible} & \end{array}$$

To prove predicate B.1, we begin with the assumption

$$\begin{array}{l} \forall a : \mathit{join}(S1, S2).\mathit{attribs} \cap S3.\mathit{attribs} \\ \quad \bullet \ \mathit{join}(S1, S2).\mathit{type}(a) = S3.\mathit{type}(a) \end{array}$$

By the definition of *join*, this is equivalent to

$$\begin{array}{l} \forall a : \mathit{join}(S1, S2).\mathit{attribs} \cap S3.\mathit{attribs} \\ \quad \bullet \ (S1.\mathit{type} \cup S2.\mathit{type})(a) = S3.\mathit{type}(a) \end{array}$$

By the first condition on the definition of *join*, we know that $S1.type$ and $S2.type$ agree on common attributes, so the above predicate implies

$$\forall\, a : S2.attribs \cap S3.attribs \bullet S2.type(a) = S3.type(a)$$

as desired.

To prove predicate B.2, we begin with the assumption

$$\exists\, s_{12} : join(S1, S2).states;\ s_3 : S3.states \bullet (s_{12}, s_3) \in compatible$$

By the definition of $join(S1, S2)$, this implies

$$\exists\, s_1 : S1.states;\ s_2 : S2.states;\ s_3 : S3.states \bullet ((s_1, s_2) \in compatible \wedge (s_1 \cup s_2, s_3) \in compatible)$$

By Lemma B.1, this implies

$$\exists\, s_2 : S2.states;\ s_3 : S3.states \bullet (s_2, s_3) \in compatible$$

as desired.

To prove predicate B.3 we must satisfy $\forall\, a : S1.attribs \cap join(S2, S3).attribs \bullet S1.type(a) = join(S2, S3).type(a)$ By the definition of $join(S2, S3)$, we know

$$join(S2, S3).attribs = S2.attribs \cup S3.attribs$$

$$join(S2, S3).type = S2.type \cup S3.type$$

We have two cases to investigate.

Case 1: $a \in S2.attribs$

Since $(S1, S2) \in \mathrm{dom}\, join$, we are done because of the assumption

$$\forall\, a : S1.attribs \cap S2.attribs \bullet S1.type(a) = S2.type(a)$$

Case 2: $a \in (S3.attribs - S2.attribs)$

Since $(join(S1, S2), S3) \in domjoin$, we know

$$\forall\, a : join(S1, S2).attribs \cap S3.attribs \bullet join(S1, S2).type(a) = S3.type(a)$$

and by the definition of *join* and the assumption that $a \notin S2.attribs$, we have

$$\begin{array}{l} \forall\, a : S1.attribs \cap S3.attribs \\ \quad \bullet \;\; S1.type(a) = S3.type(a) \end{array}$$

as desired.

To prove predicate B.4, we can use the assumptions to satisfy

$$\begin{array}{l} \exists\, s_1 : S1.states;\;\; s_2 : S2.states;\;\; s_3 : S3 \\ \quad \bullet \;\; (\quad (s_1, s_2) \in compatible \\ \qquad \wedge\, (s_1 \cup s_2, s_3) \in compatible \end{array}$$

which by Lemma B.1 and the definition of *join* gives us

$$\begin{array}{l} \exists\, s_1 : S1.states;\;\; s_{23} : join(S2, S3).states \\ \quad \bullet \;\; (s_1, s_{23}) \in compatible \end{array}$$

as desired.

We have now shown that $join(S2, S3)$ and $join(S1, join(S2, S3))$ are defined. We can now show the equality required by the theorem. Equality of the attribute sets and the type function relies on the associativity of $\cup$. To show equality of the state sets, we begin with

$$\begin{array}{rl} join(join(S1, S2), S3).states = \{ & s_{12} : join(S1, S2).states;\;\; s_3 : S3.states \\ \mid & (s_{12}, s_3) \in compatible \\ \bullet & s_{12} \cup s_3 \;\} \end{array}$$

By the definition of *join* and Lemma B.1, this is equivalent to

$$\begin{array}{rl} join(join(S1, S2), S3).states = \{ & s_1.states;\;\; s_2 : S2.states;\;\; s_3 : S3.states \\ \mid & (\quad (s_1, s_2) \in compatible \\ & \quad \wedge\, (s_2, s_3) \in compatible \\ & \quad \wedge\, (s_1 \cup s_2, s_3) \in compatible \\ & \quad \wedge\, (s_2, s_2 \cup s_3) \in compatible \\ \bullet & s_1 \cup s_2 \cup s_3 \;\} \end{array}$$

which is equivalent to $join(S1, join(S2, S3)).states$, as desired.
◇ END OF PROOF OF THEOREM 5.1

Appendix C

Detailed semantics for the agent language

In this appendix we provide a more detailed semantics for the agent language than provided in Chapter 5. The semantics is given in parts. We first define the overall semantic operator which takes whole agent expressions and maps them into a system of agents. The agent language mirrors the development of the agent model, and so we have separate sections of the language which individually treat the description of internal, communication and external specifications. Each section will have its own semantic operator which maps expressions of its language to specifications in the agent model.

Agents in the interactive system

An interactive system is a mapping from agent identifiers to the set of agents in *Agent*. We introduce a set of possible agent identifiers.

$$[AgentID]$$

$$IntSys == AgentID \pfun Agent$$

The system semantic function, $\mathcal{S}[\![\_]\!]$, takes an existing interactive system and an agent language expression and produces a new interactive system. The agent language description represents either the synchronization of existing agents, the interleaving of an existing agent with a 3-part description of a new agent (internal, external, communication specification), or a completely new 3-part description of an agent. The following is a BNF-like description of the agent language syntax.

Square brackets are used to indicate an item which is optional.

AgExp ::=	**agent** *AgentID*	– synchronization
	synchronizes *AgentIDList*	
	[**with** 3*PartSpec*]	
	endagent *AgentID*	
|	**agent** *AgentID*	– interleaving
	interleaves *AgentIDList*	
	[**with** 3*PartSpec*]	
	endagent *AgentID*	
|	**agent** *AgentID*	– 3-part specification
	3*PartSpec*	
	endagent *AgentID*	

The system semantic function $\mathcal{S}[\![\_]\!]$ is defined structurally over the elements in *AgExp*. For synchronized combination, the expression

agent *A*1
synchronizes *AS*
with *Spec*
endagent *A*1

maps the (fresh) agent identifier *A*1 to the synchronous composition of the agents indicated by the sequence of (distinct) agent identifiers *AS*, if such a composition is allowed by $composeall_{sync}$. This may then he interleaved with the agent defined by the 3-part specification *Spec*, according to the semantic operator $\mathcal{A}g[\![\_]\!]$ discussed later.

$$
\begin{array}{l}
\mathcal{S}[\![\ \_\ ]\!] : (\mathit{IntSys} \times \mathit{AgExp}) \nrightarrow \mathit{IntSys} \\
\hline
\forall\, A1 : \mathit{AgentID};\ AS : \mathrm{seq}_1\, \mathit{AgentID};\ \rho : \mathit{IntSys} \\
\quad \mid\ (\ \ A1 \notin \mathrm{dom}\, \rho \\
\qquad \wedge\ (AS \fatsemi \rho) \in \mathrm{dom}\, \mathit{composeall}_{\mathit{sync}}) \\
\quad \bullet\ \mathcal{S}[\![\left(\begin{array}{l} \textbf{agent}\ A1 \\ \textbf{synchronizes}\ AS \\ \textbf{endagent}\ A1 \end{array} \right)]\!] = \\
\qquad \rho \oplus \{ A1 \mapsto \mathit{composeall}_{\mathit{sync}}(AS \fatsemi \rho) \} \\
\forall\, A1 : \mathit{AgentID};\ AS : \mathrm{seq}_1\, \mathit{AgentID};\ \mathit{Spec} : 3\mathit{PartSpec}\ \rho : \mathit{IntSys} \\
\quad \mid\ (\ \ A1 \notin \mathrm{dom}\, \rho \\
\qquad \wedge\ AS \fatsemi \rho \in \mathrm{dom}\, \mathit{composcall}_{\mathit{sync}} \\
\qquad \wedge\ (\mathit{composeall}_{\mathit{sync}}(AS \fatsemi \rho), \mathcal{A}g[\![\ \mathit{Spec}\ ]\!]) \in \mathrm{dom}\, \mathit{compose}_{\mathit{int}}) \\
\quad \bullet\ \mathcal{S}[\!| \left(\begin{array}{l} \textbf{agent}\ A1 \\ \textbf{synchronizes}\ AS \\ \textbf{with}\ \mathit{Spec} \\ \textbf{endagent}\ A1 \end{array} \right) |\!] = \\
\qquad \rho \oplus \{ A1 \mapsto \mathit{composc}_{\mathit{int}}(\mathit{composeall}_{\mathit{sync}}(AS \fatsemi \rho), \mathcal{A}g[\![\ \mathit{Spec}\ ]\!]) \}
\end{array}
$$

Note that because of the associativity of $\mathit{compose}_{\mathit{sync}}$ (Theorem 5.5, the order of the agent identifiers in AS does not matter.

For interleaved combination, the expression

agent $A1$
interleaves AS
with *Spec*
endagent $A1$

maps the fresh identifier $A1$ to the interleaved product of the known agent definitions in AS and the 3-part specification *Spec*, if given.

$$\begin{array}{l}
\forall\, A1 : \mathit{AgentID};\ AS : \mathrm{seq}_1\ \mathit{AgentID};\ \rho : \mathit{IntSys} \\
\quad |\ (\ A1 \notin \mathrm{dom}\,\rho \\
\qquad \wedge\ AS \fatsemi \rho \in \mathrm{dom}\ \mathit{composeall}_{\mathit{int}}) \\
\quad \bullet\ \mathcal{S}[\!\left(\begin{array}{l}\textbf{agent } A1 \\ \textbf{interleaves } AS \\ \textbf{with } \mathit{Spec} \\ \textbf{endagent } A1\end{array}\right)\!] = \rho \oplus \{A1 \mapsto \mathit{composeall}_{\mathit{int}}(AS \fatsemi \rho)\}
\end{array}$$

$$\begin{array}{l}
\forall\, A1 : \mathit{AgentID};\ AS : \mathrm{seq}_1\ \mathit{AgentID};\ \mathit{Spec} : 3\mathit{PartSpec};\ \rho : \mathit{IntSys} \\
\quad |\ (\ A1 \notin \mathrm{dom}\,\rho \\
\qquad \wedge\ (AS \fatsemi \rho) \frown \mathcal{A}g[\!|\ \mathit{Spec}\ |\!] \in \mathrm{dom}\ \mathit{composeall}_{\mathit{int}}) \\
\quad \bullet\ \mathcal{S}[\!\left(\begin{array}{l}\textbf{agent } A1 \\ \textbf{interleaves } AS \\ \textbf{with } \mathit{Spec} \\ \textbf{endagent } A1\end{array}\right)\!] = \\
\qquad \rho \oplus \{A1 \mapsto \mathit{composeall}_{\mathit{int}}((AS \fatsemi \rho) \frown \mathcal{A}g[\!|\ \mathit{Spec}\ |\!])\}
\end{array}$$

Note that because of the associativity of $\mathit{compose}_{\mathit{int}}$ (Theorem 5.4, the order of the agent identifiers in AS does not matter.

A stand alone specification of an agent,

agent $A1$
Spec
endagent $A1$

maps the (fresh) identifier $A1$ to the agent $\mathcal{A}g[\!|\ \mathit{Spec}\ |\!]$.

$$\begin{array}{l}
\forall\, A1 : \mathit{AgentID};\ \mathit{Spec} : 3\mathit{PartSpec};\ \rho : \mathit{IntSys} \\
\quad |\ (\ A1 \notin \mathrm{dom}\,\rho \\
\qquad \wedge\ \mathit{Spec} \in \mathrm{dom}\ \mathcal{A}g[\!|\ _\ |\!]) \\
\quad \bullet\ \mathcal{S}[\!\left(\begin{array}{l}\textbf{agent } A1 \\ \mathit{Spec} \\ \textbf{endagent } A1\end{array}\right)\!] = \rho \oplus \{A1 \mapsto \mathcal{A}g[\!|\ \mathit{Spec}\ |\!]\}
\end{array}$$

The three part specification of an agent is given by an internal, external and communication langnage.

$$\begin{array}{rl}
3\mathit{PartSpec} ::= & \textbf{internal } \mathit{IExp} \\
& \textbf{communication } \mathit{CExp} \\
& \textbf{external } \mathit{EExp}
\end{array}$$

The agent semantic operator, $\mathcal{A}g[\!|\ \_\ |\!]$, is defined in terms of semantic operators for each of the sublanguages. The domain of $\mathcal{A}g[\!|\ \_\ |\!]$ is the set of combinations of internal, communication and external expressions which yield a valid agent description when they are mapped to their respective specifications in the model.

$$
\begin{array}{l}
\mathcal{A}g[\![\ \_\ ]\!] : 3PartSpec \nrightarrow Agent \\
\mathcal{I}[\![\ \_\ ]\!] : IExp \rightarrow InternalSpec \\
\mathcal{C}[\![\ \_\ ]\!] : CExp \rightarrow Communication \\
\mathcal{E}[\![\ \_\ ]\!] : EExp \rightarrow ExternalSpec \\
\hline
\mathrm{dom}\ \mathcal{A}g[\![\ \_\ ]\!] = \{\ IE : IExp;\ CE : CExp;\ EE : EExp;\ Agent \\
\qquad \mid\ (\ \theta InternalSpec = \mathcal{I}[\![\ IE\ ]\!] \\
\qquad\quad \land\ \theta Communication = \mathcal{C}[\![\ CE\ ]\!] \\
\qquad\quad \land\ \theta ExternalSpec = \mathcal{E}[\![\ EE\ ]\!]) \\
\qquad\quad \bullet \left(\begin{array}{l} \textbf{internal}\ IE \\ \textbf{communication}\ CE \\ \textbf{external}\ EE \end{array} \right) \} \\
\\
\mathcal{A}g[\![\left(\begin{array}{l} \textbf{internal}\ IE \\ \textbf{communication}\ CE \\ \textbf{external}\ EE \end{array} \right)]\!] = \mu\, Agent \\
\qquad\qquad\qquad\qquad \bullet\ (\ \theta InternalSpec = \mathcal{I}[\![\ IE\ ]\!] \\
\qquad\qquad\qquad\qquad\quad \land\ \theta Communication = \mathcal{C}[\![\ CE\ ]\!] \\
\qquad\qquad\qquad\qquad\quad \land\ \theta ExternalSpec = \mathcal{E}[\![\ EE\ ]\!])
\end{array}
$$

In the following subsections, we will outline the definition of the semantic operators for each of these sublanguages.

A language for internal specifications

The internal specification language is given by the following BNF-like description.

$$
\begin{array}{lll}
\mathit{ISpecLanguage} & ::= & \textbf{types}\ \mathit{Decls} \\
 & & \textbf{attributes}\ \mathit{AVmap} \\
 & & \textbf{invariant}\ \mathit{AVPred} \\
 & & \textbf{operations}\ \mathit{OpList}
\end{array}
$$

The semantic function $\mathcal{I}[\![\ \_\ ]\!]$ can be described in terms of smaller semantic functions on the different parts of *ISpecLanguage*.

Decls is a list of type declarations and constructions local to the agent definition. In the declaration, we can build up type definitions from basic sets, exactly as is done in Z. For example, we could write the type definitions for the window.

$$
\begin{array}{l}
[ICON] \\
xmax, ymax : \mathbb{N} \\
PIXEL == 0 \mathinner{..} xmax \times 0 \mathinner{..} ymax
\end{array}
$$

There is a semantic function which produces the sets in the valuespace $\mathcal{V}$ similar to the *Carrier* function in Spivey's semantics. We will also allow the definition of global types outside the definition of any agent, as is allowed in Z.

AVmap is a list of the attributes for which each state of the agent has a value. The *AVmap* is written much the same way that components declarations are written in a standard Z schema, so we could write expressions like the following in the **attributes** section of an agent.

$$icon : ICON$$
$$iconpos, winpos, winsize : PIXEL$$
$$status : open \mid closed$$

These state declarations are mapped by the semantic function type$[\![_]\!]$ to obtain the *type* information for the agent.

AVPred is a predicate on the values of the attributes which must be satisfied by all states of the agent. We allow the same kinds of predicate expressions as is possible in Z. The semantic function state$[\![_]\!]$ takes the **attributes** and the **invariants** information and yields the set of possible states for the agent.

OpList is a list describing the possible transitions for the agent. The state transitions are given pre- and postcondition semantics, along with an explicit framing conditions which lists the attributes that can be changed by the operation. Each operation definition gives a family of state transitions indexed by a message identifier along with its typed parameters. The window moving operation for an open window would be defined as follows.

operations
move_open(*newpos* : *PIXEL*)
changes *winpos*
pre *status* = *open*
post *winpos'* = *newpos*

The BNF description of *OpList* is given below. Square brackets are used to indicate an optional occurrence of an item.

[*MessageID*]

OpList := *MessageID*(*TypedParams*)
[**changes** (*AttribList*)]
[**pre** (*BeforePred*)]
[**post** (*AfterPred*)]
[*OpList*]

The operation has a name taken from the set *MessageID*. A typed parameter list, of the same form as used to declare the state attributes, lists the generic arguments for the operation. A message is formed by instantiating each argument with an

actual value. For example, a message generated from the definition above would be $move\_open((0,0))$. The **changes** clause provides an explicit framing condition for the operation definition. The precondition is a predicate on the state attributes and the arguments. The postcondition is a predicate on the before and after value of state attributes and the arguments. The values of attributes before and after the state transition are distinguished by undashed and dashed attribute identifier names, as is the convention in Z. Together, the framing, pre- and postconditions define a state transition.

The semantic function state$[\![_]\!]$ takes a list of operations in *OpList* and creates the *operations* function for the agent. Each single operation defined as above yields a family of state transition relations, indexed by the possible messages.

$$| \quad \text{state}[\![\_]\!] : \mathit{OpList} \rightarrow (\mathit{Message} \nrightarrow (\mathit{State} \leftrightarrow \mathit{State}))$$

A language for communication specification

The communication specification is straightforward, since it simply lists the input and output channels for the agent along with the messages that can be passed along those channels. Synchronized channels arise from synchronous combination of existing agents, it is not possible to explicitly declare a channel as synchronized in the notation. The burden rests with the designer to ensure channel fidelity, i.e., the specification cannot have the same agent using the same channel as input and output, nor can a synchronized channel be an input or output channel for another agent. The internal messages are also declared explicitly as associated to the τ channel. A BNF description of the syntax for communication is given below.

$$\begin{array}{lll}
\mathit{CSpecLanguage} & ::= & [\textbf{inputs}\ \mathit{TypedChannellist}] \\
 & | & [\textbf{outputs}\ \mathit{TypedChannellist}] \\
 & | & [\tau : \mathit{Messagelist}] \\
\mathit{TypedChannelList} & ::= & \mathit{ChannelID} : \mathit{Messagelist}\ [\mathit{TypedChannellist}] \\
\mathit{Messagelist} & ::= & \mathit{MessageID}(\mathit{TypedParameters})\ [, \mathit{Messagelist}]
\end{array}$$

As with the definition of **operations** _, the set of messages is generated by instantiating all of the parameters in the *TypedParameters* list with values according to their types. The semantic function $\mathcal{C}[\![\_]\!]$ takes an element in *CSpecLanguage* and produces the communications specification.

$$| \quad \mathcal{C}[\![\_]\!] : \mathit{CSpecLanguage} \nrightarrow \mathit{Communication}$$

A language for external specifications

The external specification language is developed in two parts as described in Section 5.3. In this section we will give the semantics for the constructive language

for generating external specifications of agents. First, we recall the definition of an external specification as given in Chapter 5.

$$
\begin{array}{|l}
\textit{ExternalSpec} \\
\hline
\textit{alphabet} : \mathbf{P}\ \textit{Event} \\
\textit{traces} : \mathbf{P}\ \mathrm{seq}\ \textit{alphabet} \\
\hline
\langle\,\rangle \in \textit{traces} \\
\mathrm{prefix\_closed}\ (\textit{traces}) \\
\hline
\end{array}
$$

The syntax for the constructive language follows very closely the syntax for CSP given by Hoare [82].

$$
\begin{array}{lll}
\textit{ConEExp} ::= & \mathbf{stop}\langle\!\langle \mathbf{P}\ \textit{Event} \rangle\!\rangle & \text{- deadlock} \\
\mid & \mathbf{run}\langle\!\langle \mathbf{P}\ \textit{Event} \rangle\!\rangle & \text{- the total behaviour} \\
\mid & \mathbf{skip}\langle\!\langle \mathbf{P}\ \textit{Event} \rangle\!\rangle & \text{- successful termination} \\
\mid & \langle\!\langle \textit{Event} \rangle\!\rangle \rightarrow \textit{ConEExp} & \text{- prefix composition} \\
\mid & \textit{ConEExp} \,\Box\, \textit{ConEExp} & \text{- choice composition} \\
\mid & \textit{ConEExp} \,;\, \textit{ConEExp} & \text{- sequential composition} \\
\mid & \textit{ConEExp} \parallel \textit{ConEExp} & \text{- synchronous parallel composition} \\
\mid & \textit{ConEExp} \,|\!|\, \textit{ConEExp} & \text{- asynchronous parallel composition} \\
\mid & f(\textit{ConEExp}) & \text{- process relabelling} \\
\mid & \mu\, X : \langle\!\langle \mathbf{P}\ \textit{Event} \rangle\!\rangle \bullet F(X) & \text{- guarded recursion}
\end{array}
$$

We can now give the definition of each of these syntactic constructions in terms of the external specification model. We will give two versions of the semantics for each construct. The first will be a direct denotational definition in terms of the model for external specifications. These definitions are very close to those given by Hoare. The mapping from the syntactic domain *ConEExp* to the semantic domain *ExternalSpec* is given by the function $\mathcal{E}_{con}[\![\ \_\ ]\!]$, which can be further broken down into functions for the alphabet, $\alpha[\![\ \_\ ]\!]$ and trace set, $\mathcal{T}[\![\ \_\ ]\!]$.

$$
\begin{array}{|l}
\mathcal{E}_{con}[\![\ \_\ ]\!] : \textit{ConEExp} \rightarrow \textit{ExternalSpec} \\
\alpha[\![\ \_\ ]\!] : \textit{ConEExp} \rightarrow \mathbf{P}\ \textit{Event} \\
\mathcal{T}[\![\ \_\ ]\!] : \textit{ConEExp} \rightarrow \mathbf{P}\ \mathrm{seq}\ \textit{Event} \\
\hline
(\mathcal{E}_{con}[\![\ ES\ ]\!]).\textit{alphabet} = \alpha[\![\ ES\ ]\!] \\
(\mathcal{E}_{con}[\![\ ES\ ]\!]).\textit{traces} = \mathcal{T}[\![\ ES\ ]\!]
\end{array}
$$

The second version of semantics will be an operational semantics which may make it easier to understand the functionality of the different constructs. This operational semantics is given by inference rules in the manner suggested by Plotkin [130] and is the standard semantics usually given for a process algebra in the CCS

family [109, 110]. We will not extend this operational semantics to show that it matches some intuitive equational theory on expressions in *ConEExp*, as our purpose for presenting it is only to increase understanding.

Deadlock

$\mathbf{stop}_A$ represents deadlock in which participation in any events from the alphabet is prohibited.

$$\begin{array}{|l} \mathbf{stop}\_ : \mathbf{P}_1\, \mathit{Event} \to \mathit{ConEExp} \\ \hline \alpha[\![\, \mathbf{stop}_A \,]\!] = A \\ \mathcal{T}[\![\, \mathbf{stop}_A \,]\!] = \{\, \langle\rangle \,\} \end{array}$$

There is no inference rule to describe the action of $\mathbf{stop}_A$ since it has no action.

Total behaviour

Given a set of events A, the expression $\mathbf{run}_A$ denotes the external specification with alphabet A which can participate in any sequence of events from A.

$$\begin{array}{|l} \mathbf{run}\_ : \mathbf{P}_1\, \mathit{Event} \to \mathit{ConEExp} \\ \hline \alpha[\![\, \mathbf{run}_A \,]\!] = A \\ \mathcal{T}[\![\, \mathbf{run}_A \,]\!] = \mathrm{seq}\, A \end{array}$$

Operationally, $\mathbf{run}_A$ participates in any event in A and then continues to behave as $\mathbf{run}_A$.

$$\frac{e \in A}{\mathbf{run}_A \overset{e}{\rightsquigarrow} \mathbf{run}_A}$$

Successful termination

We introduce a primitive construct, $\mathbf{skip}_A$ to represent the successful termination of an external specification with alphabet A. To do this, we must also introduce a special internal event, $\surd$ (read "tick") which signals the termination. This event carries no message of significance, so we label it the *null* message. It occurs along the internal channel τ. Furthermore, we constrain the external specifications set so that the $\surd$ event can only appear at the end of a trace.

$$
\begin{array}{l}
null : Message \\
\surd : Event \\
\mathbf{skip}\_ : \mathbf{P}_1\, Event \rightarrow ConEExp \\
\hline
\surd.channel = \tau \\
\surd.message = null \\
\forall\, ES : ExternalSpec;\ t : ES.traces \\
\quad \bullet\ \surd \notin \mathrm{ran}(front(t)) \\
\alpha[\![\ \mathbf{skip}_A\ ]\!] = A \cup \{\ \surd\ \} \\
\mathcal{T}[\![\ \mathbf{skip}_A\ ]\!] = \{\ \langle\,\rangle, \langle \surd \rangle\ \}
\end{array}
$$

The operational semantics of $\mathbf{skip}_A$ is given by one simple inference rule. $\mathbf{skip}_A$ can participate in the event $\surd$ after which it deadlocks.

$$\frac{}{\mathbf{skip}_A \overset{\surd}{\rightsquigarrow} \mathbf{stop}_A}$$

Prefix composition

$e \rightarrow P$ first engages in the event e and then behaves like P. The constraint on this construction is that e must already be in the alphabet of P.

$$
\begin{array}{l}
\_ \rightarrow \_ : (Event \times ConEExp) \nrightarrow ConEExp \\
\hline
\mathrm{dom}(\_ \rightarrow \_) = \{\ e : Event;\ P : ConEExp \\
\qquad\qquad \mid\ e \in \alpha[\![\ P\ ]\!] \\
\qquad\qquad \bullet\ (e, P)\ \} \\
\alpha[\![\ e \rightarrow P\ ]\!] = \alpha[\![\ P\ ]\!] \\
\mathcal{T}[\![\ e \rightarrow P\ ]\!] = \{\ \langle\,\rangle\ \} \cup \{\ t : \mathcal{T}[\![\ P\ ]\!] \bullet \langle e \rangle \frown t\ \}
\end{array}
$$

The operational semantics for prefix composition is also covered by one simple inference rule.

$$\frac{e \in \alpha[\![\ P\ ]\!]}{e \rightarrow P \overset{e}{\rightsquigarrow} P}$$

Choice composition

Choice between two external specifications, written $P \,\Box\, Q$, indicates that the behaviour can either proceed as described by P or by Q. The choice is made by

the first event in which $P \,\Box\, Q$ participates. Hoare refers to this choice operator as deterministic (or external) choice and distinguishes it from a nondeterministic choice, $P \sqcap Q$. In deterministic choice, the choice can be resolved (externally) by the environment which interacts with $P \,\Box\, Q$, whereas in nondeterministic choice the environment can have no effect. In the traces model, no distinction can be made between these processes, and so we only describe external choice.

$$\begin{array}{|l} \_\,\Box\,\_ : (\mathit{ConEExp} \times \mathit{ConEExp}) \to \mathit{ConEExp} \\ \hline \alpha[\![P \,\Box\, Q]\!] = \alpha[\![P]\!] \cup \alpha[\![Q]\!] \\ \mathcal{T}[\![P \,\Box\, Q]\!] = \mathcal{T}[\![P]\!] \cup \mathcal{T}[\![Q]\!] \end{array}$$

The operational semantics is given by two inference rules, indicating that $P \,\Box\, Q$ can proceed if one of P or Q or both can proceed. In the case where both can proceed, the choice is nondeterministic.

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P \,\Box\, Q \overset{e}{\rightsquigarrow} P'}$$

$$\frac{Q \overset{e}{\rightsquigarrow} Q'}{P \,\Box\, Q \overset{e}{\rightsquigarrow} Q'}$$

Sequential composition

$P \fatsemi Q$ behaves like P until successful termination, marked by participation in the special event $\surd$. After successful termination, it behaves like Q. Unlike CSP, we do not hide participation in the event $\surd$ in the definition of the external specification.

$$\begin{array}{|l} \_ \fatsemi \_ : (\mathit{ConEExp} \times \mathit{ConEExp}) \to \mathit{ConEExp} \\ \hline \alpha[\![P \fatsemi Q]\!] = \alpha[\![P]\!] \cup \alpha[\![Q]\!] \\ \mathcal{T}[\![P \fatsemi Q]\!] = \mathcal{T}[\![P]\!] \\ \qquad\qquad\qquad \cup \\ \qquad\qquad\qquad \{\, t : \mathcal{T}[\![P]\!], t' : \mathcal{T}[\![Q]\!] \mid \mathit{last}(t) = \surd \bullet t \frown t' \,\} \end{array}$$

Operationally, if P can successfully terminate by engaging in the event $\surd$, then $P \fatsemi Q$ can proceed to behave as Q.

$$\frac{P \overset{\surd}{\rightsquigarrow} \mathbf{stop}_A}{P \fatsemi Q \overset{\surd}{\rightsquigarrow} Q}$$

Synchronous parallel composition

There are two versions of parallel composition which we allow. The first, written as $P \| Q$ demands that P and Q synchronize on participation in events of their common alphabet. In the model description, the alphabets are combiued. The traces of the synchronized combination are those which when filtered by the alphabet of P (respectively, Q) are a legal trace of P (respectively, Q).

$$
\begin{array}{|l}
\_ \| \_ : (ConEExp \times ConEExp) \rightarrow ConEExp \\
\hline
\alpha[\![P \| Q]\!] = \alpha[\![P]\!] \cup \alpha[\![Q]\!] \\
\mathcal{T}[\![(P \| Q]\!] = \{\ t : \mathrm{seq}\, \alpha[\![P \| Q]\!] \\
\qquad\qquad \mid\ (\ t \overset{\mathrm{seq}}{\rhd} \alpha[\![P]\!] \in \mathcal{T}[\![P]\!] \\
\qquad\qquad\quad \wedge\ t \overset{\mathrm{seq}}{\rhd} \alpha[\![Q]\!] \in \mathcal{T}[\![Q]\!]) \\
\qquad\qquad \bullet\ t\ \}
\end{array}
$$

The operational semantics for $P \| Q$ indicates that both P and Q can independently participate in events private to themselves, but both mnst evolve simultaneously on events which they share. Three inference rules sum this behaviour up.

$$
\frac{P \overset{e}{\rightsquigarrow} P'}{P \| Q \overset{e}{\rightsquigarrow} P' \| Q} \quad [\ e \in alphabetP - alphabetQ\ ]
$$

$$
\frac{Q \overset{e}{\rightsquigarrow} Q'}{P \| Q \overset{e}{\rightsquigarrow} P \| Q'} \quad [\ e \in Q.alphabet - P.alphabet\ ]
$$

$$
\frac{P \overset{e}{\rightsquigarrow} P' \qquad Q \overset{e}{\rightsquigarrow} Q'}{P \| Q \overset{e}{\rightsquigarrow} P' \| Q'} \quad [\ e \in P.alphabet \cap Q.alphabet\ ]
$$

Asynchronous parallel composition

The other version of parallel composition is asynchronous, which we write as $P \| Q$. Both P and Q can evolve independently, regardless of whether they share events. The alphabet is again the union of the component alphabets. Traces are obtained by interleaving traces of the components.

$$
\begin{array}{|l}
\_ \| \_ : (ConEExp \times ConEExp) \rightarrow ConEExp \\
\hline
\alpha[\![P \| Q]\!] = \alpha[\![P]\!] \cup \alpha[\![Q]\!] \\
\mathcal{T}[\![P \| Q]\!] = \{\ s : \mathrm{seq}\, \alpha[\![P \| Q]\!] \\
\qquad\qquad \mid\ \exists t : \mathcal{T}[\![P]\!];\ t' : \mathcal{T}[\![Q]\!] \bullet s\ interleaves\ (t, t') \\
\qquad\qquad \bullet\ s\ \}
\end{array}
$$

The relation *interleaves* is defined in Appendix A.

Operationally, $P \,[\!|\, Q$ is described by two rules similar to the first two rules of $P \parallel Q$.

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P \,[\!|\, Q \overset{e}{\rightsquigarrow} P' \,[\!|\, Q}$$

$$\frac{Q \overset{e}{\rightsquigarrow} Q'}{P \,[\!|\, Q \overset{e}{\rightsquigarrow} P \,[\!|\, Q'}$$

Process relabelling

Sometimes it will be convenient to identify an external specification with a previous one with an appropriate change in the event names. The restriction on such a relabelling is that the mapping to new event names be injective which is total when restricted to the alphabet of the original external specification, so that each old event name is mapped to a unique new event name. In addition, $\surd$ must be mapped to itself. The new external specification then behaves exactly as the old, with the new event names substituted for the old.

$$\begin{array}{|l}
_(_) : ((Event \rightarrowtail\!\!\!\!\!\!\rightarrow\; Event) \times ConEExp) \nrightarrow ConEExp \\
\hline
\mathrm{dom}\,_(_) = \{\, f : Event \rightarrowtail\!\!\!\!\!\!\rightarrow\; Event;\ ES : ConEExp \\
\qquad\qquad \mid\ (\ f \in \alpha[\![\ ES\ ]\!] \rightarrowtail Event \\
\qquad\qquad\quad \wedge f(\surd) = \surd) \\
\qquad\qquad \bullet\ (f, ES)\ \} \\
\alpha[\![\ f(P)\ ]\!] = f(\!|\alpha[\![\ P\ ]\!]|) \\
\mathcal{T}[\![\ f(P)\ ]\!] = \{\ t : \mathcal{T}[\![\ P\ ]\!] \bullet t \fatsemi f\ \}
\end{array}$$

Note that in the definition of $\mathcal{T}[\![\ f(P)\ ]\!]$, we use the symbol $\fatsemi$ to represent the standard Z forward functional composition, not sequential composition of external specifications.

Recursion

We will allow a simple form of recursion which is uniquely defined using a partial order semantics (see Hoare [82, Section 2.8] and Stoy [156]). The type *ConEExp* can be considered a complete partial order using the following ordering relation.

$$\begin{array}{|l}
_ \sqsubseteq _ : ConEExp \leftrightarrow ConEExp \\
\hline
ES \sqsubseteq ES' \Leftrightarrow (\ \alpha[\![\ ES\ ]\!] = \alpha[\![\ ES'\ ]\!] \\
\qquad\qquad \wedge\ \mathcal{T}[\![\ ES\ ]\!] \subseteq \mathcal{T}[\![\ ES'\ ]\!])
\end{array}$$

In this partial order, we can define the least upper bound (*lub*) of two constructed external specifications with the same alphabet by forming the trace set from the union of the individual trace sets. We can extend this to obtain lubs for *chains* of specifications as well. Therefore, *ConEExp* is a complete partial order (with respect to the alphabet A), on which all guarded continuous functions $F : ConEExp \rightarrow ConEExp$ have a unique fixed point solution to the recursive equation $\mu X : A \bullet F(X)$, which is the lub of the infinite chain of successive applications of F to $\mathbf{stop}_A$, the bottom element in the complete partial order. We have purposely designed *ConEExp* so that all of the constructors are guarded and continuous, so the recursive equation will always have a uniqne solution.

$$
\begin{array}{|l}
chain : \mathbf{P}(\text{seq } ConEExp) \\
lub : \text{seq } ConEExp \nrightarrow ConEExp \\
\mu X : \_ \bullet \_(X) : (\mathbf{P}_1\ Event \times (ConEExp \rightarrow ConEExp)) \rightarrow ConEExp \\
\hline
ESS \in chain \Leftrightarrow \forall i : 1..(\#chain - 1) \bullet ESS(i) \sqsubseteq ESS(I + 1) \\
\text{dom } lub = chain \\
\alpha[\![\ lub(ESS)\ ]\!] = \alpha[\![\ ESS(1)\ ]\!] \\
\mathcal{T}[\![\ lub(ESS)\ ]\!] = \bigcup_{1..\#ESS} \mathcal{T}[\![\ ESS\ ]\!] \\
\alpha[\![\ \mu X : A \bullet F(X)\ ]\!] = A \\
\mathcal{T}[\![\ \mu X : A \bullet F(X)\ ]\!] = \mathcal{T}[\![\ lub(\langle F^0(\mathbf{stop}_A), F^1(\mathbf{stop}_A), \cdots \rangle)\ ]\!]
\end{array}
$$

Though the definition of recursion depends on the alphabet, in practice it is not indicated when the context of use makes clear what alphabet is intended.