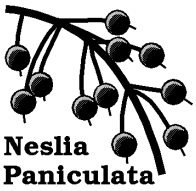
Development and Specification of Virtual Environments

Boris van Schooten

Ph.D. dissertation committee:
prof.dr.ir. A. Nijholt (promotor)
dr. E.M.A.G. van Dijk (supervisor)
dr. J. Zwiers
prof.dr. H. Zijm
prof.dr. M. Aksit
prof.dr. A. Dix
prof.dr. F.M.G. de Jong
prof.dr. E.H.L. Aarts
prof.dr. D. Konstantas
prof.dr. P.M.G. Apers (chairman)



Taaluitgeverij Neslia Paniculata
Uitgeverij voor Lezers en Schrijvers van Talige Boeken
Nieuwe Schoolweg 28, 7514 CG Enschede,
The Netherlands

CTIT Ph.D. Thesis series No. 03-47
Center for Telematics and Information Technology
P.O. Box 217, 7500 AE Enschede, The Netherlands

SIKS Dissertation Series No. 2003-06
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 90-75296-06-1

ISSN 1381-3617 (CTIT Ph.D. Thesis No. 03-47)

Enschede, The Netherlands.

The cover design shows a visualisation of the program code of the VETk toolkit.

Printed by: Printpartners Ipskamp, Enschede.

DEVELOPMENT AND SPECIFICATION OF VIRTUAL ENVIRONMENTS

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof.dr. F.A. van Vught,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, April 17, at 16:45.

by

Boris Wessel van Schooten
born on August 7, 1972
in Almelo, The Netherlands

This dissertation is approved by:
prof.dr.ir. A. Nijholt.
Assistant promotor:
dr. E.M.A.G. van Dijk.

Acknowledgements

This Ph.D. thesis concludes five years of my work as a Ph.D. student at the university of Twente. I wish to thank various people who have provided valuable help.

First of all I would like to thank Betsy van Dijk and Anton Nijholt for their continuous support throughout the Ph.D. project, and making the Parlevink group a good home for doing research.

Thanks to Job Zwiers for various comments concerning formal specifications and help on working out some of the specifications, and to Job Zwiers and Olaf Donk for a joint venture into the application of several specification languages.

Acknowledgements go to the people who were willing to join my dissertation committee, Alan Dix, Dimitri Konstantas, Emile Aarts, Mehmet Aksit, and Franciska de Jong. I am honored.

More thanks go to several people who were so kind to comment on various versions of my thesis and other writings: Rieks op den Akker, Dirk Heylen, Mannes Poel, Franciska de Jong, and Mehmet Aksit.

Last but not least, thanks to my parents for always supporting me in what I do, to Servaas Sprong for various philosophical discussions on the subject (amongst others), and Monique van Ommen and Reinoud Zandijk for their never-ending patience and moral support.

Contents

Part I

Approaches to development

Chapter 1

Introduction

1.1 The problem domain

A novel class of user interfaces, called *virtual environments* (VEs), has been emerging recently. VEs have been used for a variety of applications, such as information browsing and visualisation, simulation, tutoring, and human communication. They have also been proposed as a novel replacement of some traditional user interfaces, with the goals of improving human-computer interaction aspects such as usability or learnability.

We will now define more precisely what we mean by 'virtual environment'. The basic definition is:

A virtual environment is an interactive system that is designed in analogy with the physical world.

Usually, people associate the term with a computer imitation of a world that you can walk through. The world is often portrayed in a realistic manner, such as a 3-D *first-person perspective*, which means the world is displayed as though it is seen through the eyes of the user walking through it. However, it seems reasonable to have a broader conception of VE, which is what we will do here. We will argue that graphical user interfaces (GUIs), multimodal systems, interface agents, and multi-user systems all contain natural elements of the 'VE' concept.

We consider traditional graphical user interfaces (GUIs) a kind of VE, since they are clearly made up out of analogies of things found in the physical world: a desktop, objects that can be dragged, machines that can be manipulated by means of buttons and knobs. In practice too, many VEs are extensions of the GUI concept. We will call this *graphical interaction*. In addition to GUI-type graphical interaction, many VEs use advanced graphical techniques, such as 3D representation and animation.

Also characteristic of VEs is embodiment. Often it is the case that users are, in some way, made part of the environment. In many VEs, users have some kind of embodiment or presence, which plays a non-trivial role within the environment. Regular GUIs have little or no presence, though one might say that the mouse cursor is a form of presence:

an embodiment of the user's moving hand. We already find more presence in multi-user GUIs, where the users can often see some of the user interface elements of other users so they have some awareness of what the others are doing. For example, users may see each other's mouse cursors, which takes the role of presence that the mouse cursor plays much further. Going even further, the user might have an embodiment in the form of a depiction of a human body. This depiction is usually called an *avatar*. Note that embodiment is closely linked to the idea of using this embodiment for communication, and we find that VEs are often multi-user systems.

Conversely, the computer too may be given a body inside a VE. For example, some help systems take the form of a conversational agent, embodied in an animated character. This style of user interface is called *interface agent*. The ideal conversational agent would be able to use and understand some natural language, and thus speak with the user using his/her natural mode of communication.

Such interface agents often communicate through a combination of communication channels, such as speech, gestures, and facial expression. Systems that provide such communication through multiple channels are called *multimodal* systems. In some multimodal systems, the user can use multimodality to communicate with the system as well. Multimodality is also useful for enabling users to communicate with each other. Though there are many multimodal systems that are not VEs, we may say that VEs are often multimodal systems.

A more precise characterisation of the properties and features inherent in these different types of user interface is found in chapter 2. We will focus on the issues of developing VEs comprising these specific properties. Given that we have the goal of building a specific VE for a specific application area, we can now ask ourselves how this is best done. In this context, we can define the main research question:

How is a virtual environment best developed?

To answer this question, we will look at the solutions provided by the literature and existing software toolkits. For each different property, a variety of design issues may be identified, both from the users' perspective (what design is best for the user?), and from the technical perspective (what design is the least difficult to build?). Then, we will propose a first version of a development technique of our own, which starts with a set of specification languages.

Although limited VEs such as GUIs have been built with some success for a relatively long time, more 'state of the art' VEs are more difficult to develop. We will give some examples of state of the art VEs in chapter 2. Development of 'state of the art' VEs is an entire research project by itself, and there is little in the way of development frameworks to produce them. We will argue that this requires an integrated approach that facilitates the development of all of the different properties and features of these VEs. We will work towards our research goal in the following manner:

- Provide an overview of the issues of VE development.
- Provide an assessment of the different means that have been devised to address them.

- Provide a first version of a development framework based on this information.

This is still quite a general goal statement, and as we shall see, the work soon focusses upon much more specific issues. Most of the thesis will be concerned with specification languages and techniques, and will eventually lead to a specification technique of our own. Nevertheless, it is one of our aims not to lose sight of the broader issues, so that the context in which the specification technique will be used is made clearer. While a specification technique is only part of a development framework, other parts of the framework may be more precisely examined in future research.

1.2 Overview of issues and solutions

We have begun by characterising VEs as having a number of specific properties: graphical interaction, multimodal interaction, interface agents, and multi-user. Each aspect is found in different classes of interactive applications, but in VEs, most of these properties are present. Each of these aspects has specific technical and usability issues. There is a diverse body of literature that proposes solutions to them. Solutions are found in the form of development methodologies, analysis methods, design theory, and design specification. We will give an overview of these, and will focus on design specification. A number of specification languages will be reviewed, and, based on this knowledge, a new specification technique consisting of several integrated languages is proposed. The specification technique is illustrated with some examples, and further development of the technique and integration with the other aspects of development will be discussed in the conclusion. We will now give a short overview of the issues and solutions.

In graphical systems, the user interface has a complex spatial and temporal structure. Its complexity also poses significant technical issues. How should its visual layout, behaviour, and time evolution be designed and evaluated for usability? Existing specification models of both GUIs and 3D UIs are typically centred around the (graphical) structure of the environment, so that the correspondence of a specification with its appearance to the user is clear. Coupled with this structure is typically a data propagation model that models the difficult issue of keeping graphics mutually consistent and consistent with the information they are trying to display.

Interface agents and multimodality imply more complexity. Enabling a natural language conversation at a reasonable level is known to be a difficult artificial intelligence (AI) problem. With multimodality, some parts of the system have to gather and interpret, or output information in an integrated manner. How is a multimodal interface agent designed and evaluated? What requirements does the need for access to complex information pose on the software architecture? Natural language and multimodal interpretation requires a heuristic interpretation mechanism, which is often designed using iterative development, but which may benefit from high-level programming language support.

Many VEs are multi-user applications. What usability issues arise when multiple users are participating simultaneously? The presence of multiple users implies a distributed system. Issues related to distributed systems impose some quite difficult technical

constraints. How do these issues impact system design and the system development process? Often, models made for designing multi-user software include solutions to distributed systems issues, of which there are various. Visibility and communication requirements of the users place extra architectural requirements on the system, which may be satisfied in various ways.

These different issues and existing solutions have some overlap which is worth examining when creating a development framework to encompass them all. For example, interface agents and multi-user systems have many things in common, and so do task modelling and interface agent behaviour modelling. One of the outstanding issues is how to combine existing techniques effectively to build complex VEs systematically. Another is to fill in the gaps not satisfactorily covered by existing solutions, such as high-level behaviour specifications of animated objects, design techniques for 3D spaces, integrating interface agents with other kinds of interaction, and development techniques for natural language speaking and multimodal agents situated in a VE. We will give a more or less complete overview of existing issues and techniques before we propose our own development technique.

1.3 VE development and specification

We describe the development process as an iterative design-analysis cycle, with as input system specifications, and a selection of users and settings, and as output feedback information. The output may be obtained in various ways, ranging from specification walkthrough to formal verification to full usability testing. We identify a number of different kinds of input and output, for each of the different aspects of VEs. From this overview, we arrive at a set of requirements for specification languages. In particular, we will emphasise readability to non software engineers, the ability to generate prototypes, the ability to do some form of automatic verification, and a well-defined correspondence of each specification with the final system.

Since we define specification as any kind of descriptions that are used as input into the design-analysis cycle, we will not just consider 'formal specification' or 'requirement specification', but rather, *any* description that describes aspects of the subject at hand. There may be specifications which are not noted as such but which do form a part of the description of the subject. For example, when trying to understand a subject people have a natural tendency to scribble down personal notes in the form of graphs, tables, or text. By identifying and formalising such 'informal' specification processes, we may be able to capture more of the essence of what systems development is about.

In this text, we will primarily be concerned with specifying the VE software. It is argued that, usually, multiple languages are required and used. In this context, we can more precisely define the term *specification technique*, as it is used in the rest of the text.

A specification technique is a set of interrelated specification languages, meant to specify the relevant aspects of a specific subject.

After identifying a set of specification language requirements, we review a number of

existing specification languages, give an overview how they have been used for VEs, and give examples of specifications. The languages that we will review are predicate logic based languages, logic programming, temporal and intentional logic, constraint programming, structure diagrams, dataflow diagrams and component ('glue') languages, Petri nets, state automata, and process algebras.

We arrive at a specification technique that is centred around modelling the user interface as a data structure. It combines structure diagrams, assertion checking based on the Object Constraint Language, with a communication scheme based on queries on this data structure using a lightweight database. We illustrate the usage of the technique with examples, and discuss its usefulness and future improvements. We will conclude with an overview of what has been achieved and what remains to be done.

1.4 Structure of the text

The text consists of three parts. Part I classifies and assesses existing approaches to VE development, putting the main topic of this thesis, specification, in its proper context. Part II classifies and assesses a number of existing approaches to specification. Part III describes the specification technique that we have developed and evaluates it.

This text is written for people who:

- are looking for an entry point into VE development and related literature. The main parts of interest are parts I and II.
- are interested in trying out the VETk specification technique proposed here. The main part of interest is part III, which may serve as a tutorial of the technique, and discusses example applications.
- wish to continue research in the directions set out in this work. All parts and particularly the conclusions are of interest here. The conclusions show which items are unfinished and planned as (near) future research.

Each chapter is mostly self-contained, and gives the most important references to previous chapters, so the impatient reader may skip to any desired chapter. Figure 1.1 gives an overview of the areas covered by the different chapters of part I and II, and shows how the research topic is refined towards specific types of specification languages. Figure 1.2 gives an overview of the relation between the topics covered in part I and II, and our approach to specification.

Chapter 2 gives a characterisation of virtual environments by looking at existing interactive software and identifying a number of common properties. Chapter 3 provides an overview of the human-computer systems development process and some of its most common activities. Chapter 4 discusses some theoretical principles concerning human-computer systems, looking in particular at the properties identified in chapter 2. Chapter 5 discusses the properties that a specification language should have, in the light of the previous chapters. Chapter 6 discuss various specification languages. Chapter 7 and 8 discuss the specification technique proposed here. Chapter 9 concludes the text.

Some of the content of part I and II was already featured in two technical reports, (van Schooten, 1999), which mainly concerns part I, and (van Schooten, 2000a), which mainly concerns part II.

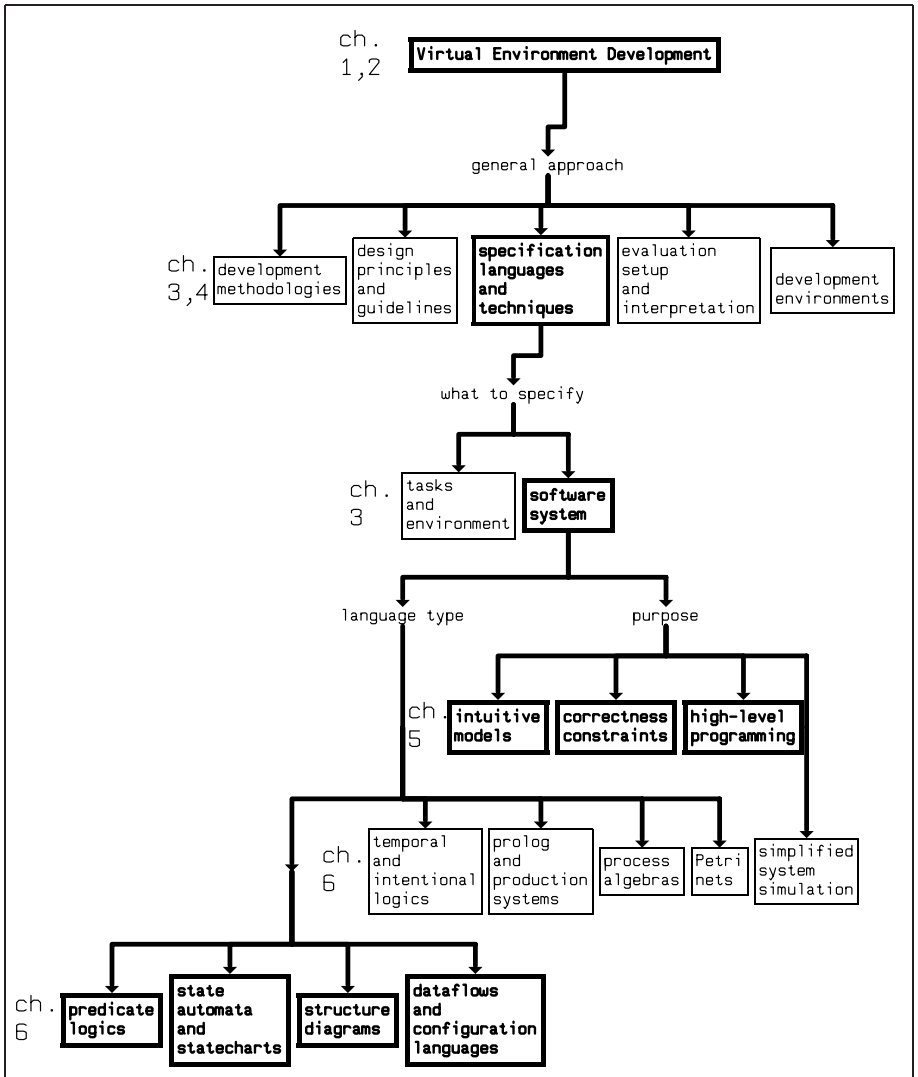


Figure 1.1: Tree representing refinement of research emphasis
A general classification of the problem domain. The items emphasised in this thesis are in boldface. This figure corresponds to the contents of part I and II of the text.

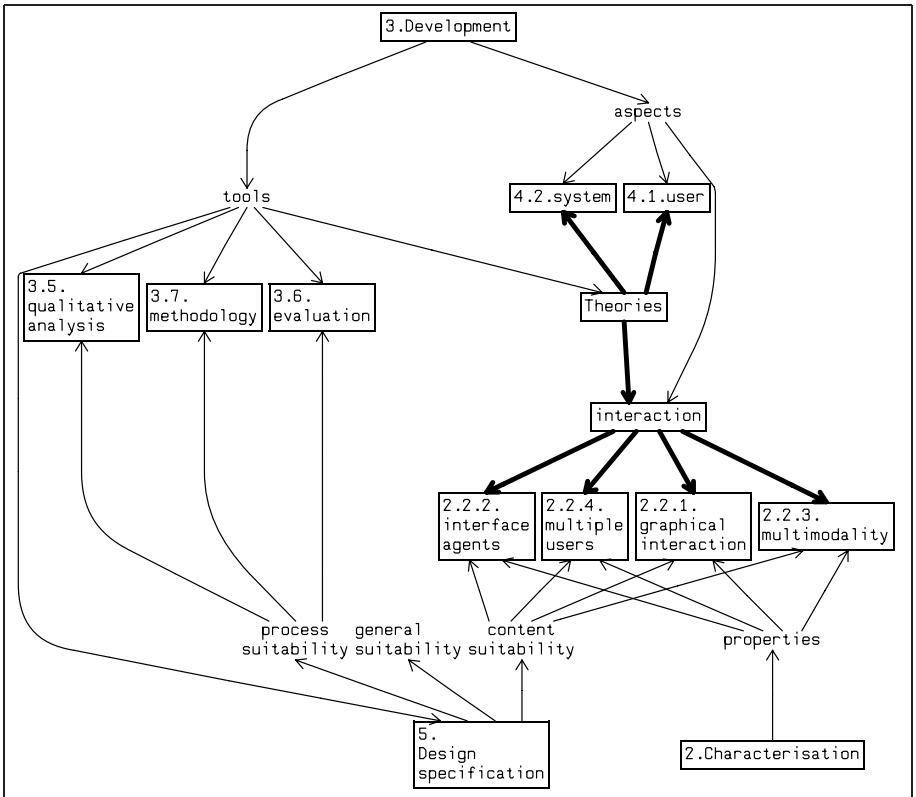


Figure 1.2: Conceptual graph of relation between topics covered in chapters 1-5.

This graph illustrates the main line of reasoning that is followed in chapter 1-5. There are several classifications that are defined in one chapter and then used in subsequent chapter(s). This graph is meant to clarify the structure of the text with respect to these classifications.

The boxes indicate topics, with some topics being explained in particular sections. The words without surrounding boxes indicate classifications according to a given criterium. The different elements of the classification are referred to using thin arrows. The thick arrows indicate composition: the child topic is a subtopic of the parent topic.

Chapter 2

The characteristics of virtual environments

2.1 Introduction

The more traditional concept of VE is that of a 3D first person graphical environment, but we argued in chapter 1 that there are several other features that may naturally be considered characteristic of VEs. In this section, we will further clarify what kinds of human-computer systems we consider VEs.

We characterised VEs by the following properties or characteristics: graphical interaction, multimodality, presence of conversational agents, and multi-user. Various types of application do have some of these properties, even though many have never been characterised as VEs. Others have been characterised as VEs, but only have a limited number of properties of them. We will however emphasise state-of-the-art VEs, VEs which have almost all of the properties. The most well known state-of-the-art VEs will be discussed at the end of the chapter.

Each property is prominent in specific systems, and has its own set of specific problems and body of literature. We will discuss the properties along with the systems in which they are most prominent in the following sections. A summary of the properties, and a few subdivisions into sub-properties that we will use to classify the different applications, are found in figure 2.2.

If we look at overviews of HCI, such as (Shneiderman, 1998), (Dix et al., 1998), (Veer and Lenting, 1995), and (Preece et al., 1994), we find some related classifications. One is the classification into general approaches to interaction, called *interaction styles* or *interaction paradigms*. Typically distinguished are interface agents as an interaction style, sometimes called *conversational interfaces*. Dix et al. distinguishes a related style, *agent-based interfaces*, in which 'autonomous software agents' can be given relatively complex instructions, so that complex tasks can be delegated to them. Graphical interaction is typically called *direct manipulation*, which usually means any kind

of point-and-click style of interaction with immediately visible feedback, i.e. GUIs. Shneiderman identifies the concept *virtual reality* as a kind of direct manipulation, a much narrower definition than our own, mainly focussing on the graphical aspect of VEs. Dix et al. treats virtual reality as a separate type of UI, again focussing on the graphical aspect. Multimodal interfaces are generally little covered in introductory books, though Dix et al. identifies this as an interaction style. Multi-user UIs are typically considered a separate kind of human-computer system. We will include some basic theory on these interaction styles along with the description of the properties in the next section.

Note that regular input and output devices are not always considered sufficient for VE interaction, and alternative devices are designed specifically for VEs, such as data gloves, head position tracking devices, and head-mounted displays. In this text, we will not emphasise such special devices, and mostly limit ourselves to either interaction using regular personal computer hardware (also known as *desktop VR*), or to abstracting away from the precise interaction devices used.

2.2 The characteristics of VEs

What follows is a coverage of the identified properties, along with various types of applications that have these properties. The types of relevant applications that we will distinguish in fact cover a significant part of the applications that have been built over the years. This is not meant as a classification into application types, but rather, it serves as a more or less complete illustration of the possibilities.

2.2.1 Graphical interaction

The ‘GUI’ was originally conceived as a digital office desktop: a metaphor which should make applications easier to understand for office workers. In practice, there are many things about GUIs that step outside of the metaphor, and ‘GUI’ actually stands for a variety of interaction techniques that enable effective use of a graphical display for interaction.

GUIs have been made widely available with the introduction of the Apple Macintosh in 1984. While there are many different GUI systems, each with a different appearance, they have a lot in common if you look at the basic interaction techniques they use. Most have for example windows, text forms, buttons and icons, menus and lists, sliders and scrollbars, text selection, and drag-and-drop. This corresponds to Shneiderman’s well known classification of GUI techniques, (Shneiderman, 1998), who distinguishes direct manipulation, menu selection, and form fill-in, and dialog boxes.

Next to this, there is a rich variety of other techniques, most of them being useful for specific types of applications. In conferences such as CHI, dozens of new techniques or improvements over older techniques are proposed each year. Most GUIs, however, use much the same interaction techniques, which have evolved only slowly. Possibly, at a certain point, familiarity becomes as important as other aspects of usability, and famil-

iar UIs techniques are preferred over improved ones. Yet, some of the GUI applications that we have now are among the most complex and difficult applications to build. Some advanced tools are now available to the developer, enabling creation of simple GUI applications relatively easily. However, development of more complex GUIs remains a difficult enterprise, especially when non-standard interaction techniques are required, such as manipulation of objects on the canvas in a drawing tool.

Generally, most work on HCI focusses on GUIs. There is a large body of literature on GUI display and interaction techniques. One of the most basic advantages of the GUI is that it makes some things simpler for the beginner than some existing alternatives, such as command languages. They are easier to learn and remember because, by the effective use of a graphical display, it is often possible to make the range of possible options more easily visible, and the results of many actions are also immediately visible. Additionally, the graphical nature of the interaction makes working with information that is essentially graphical (such as diagrams) easier.

Web interaction

A newer variant of the GUI is the Web-based user interface. We define a Web-based UI as one that interfaces with a Web site, and uses a Web browser (that is, a browser that is like the more advanced browsers now commonly available) as its technical medium. While the main goal of Web-based UIs remains to browse through (mostly textual) information, there are now various Web sites that offer interactive features for doing various tasks: communication, personalised information retrieval, banking, software management, shopping, and games. Some Web sites also include multi-user features, such as means of communication between users.

In some cases, the user interfaces are complex, and take the current technical possibilities to their limits. In fact, the Web and the Web browser are being used for things that they are not originally designed for. While the Web is full of ad-hoc standards and solutions which have emerged by demand, the interoperability problems that have to be addressed in the Web are in fact quite formidable. Consequently, Web-based applications are infamous for their unreliability (Dillman et al., 1998).

Nevertheless, Web-based UIs illustrate some interesting new concepts. Most are built in the style of GUIs, and have adopted many features of them. However, most of them are limited to form filling and list selection. Graphical interaction with direct feedback, such as drag-and-drop, is seldom found. On the other hand, Web-based user interfaces provide new techniques not found in GUIs. Particularly notable is the concept of navigation: following hyperlinks, and managing histories and bookmarks. Another is the approach of using text layout techniques: plain text, tables, pictures, and user interface elements can easily be combined within the same layout system. Since there are so many similarities between GUIs and web browsers, we have seen increased integration of the Web browser with the rest of the GUI. Help information is accessed through a Web browser, and a Web interface is sometimes used instead of a regular GUI.

Web interaction has received increased attention in the literature. Introductory books

(Dix et al., 1998; Shneiderman, 1998) now include Web interaction as well. It is usually referred to as *hypertext* or *hypermedia*, emphasising the aspect of navigation. Guidelines are given on ways to present information with help of hyperlinks, and about navigation through them.

On the technical side, Web interaction has changed the way we look at distributed software systems. Distributed systems have their own set of issues and theories. Web-based technologies have had to provide solutions to some distributed systems problems, and provide a framework to build distributed applications in. In particular, the relatively well-decoupled Web client / Web server scheme forms a new solution to distributed software systems issues. A particularly interesting related problem is that of software interoperability: Web interaction has increased the attention paid to interoperability issues.

3D graphical environments

A prime example of a 3D graphical environment is the modern computer game. But games have more characteristics of VEs than just the graphical interaction. The player can walk around in a realistic 3D environment, and interact with both 'virtual' objects and various types of autonomous (though usually not too intelligent) agents. One can also say that even the oldest video games were clear examples of VEs, save for the graphical realism. Games have been well established longer than any other kind of VE. While they are meant for entertainment and do not have any external goal to achieve, significant parts of the games' interfaces are goal-based, having the purpose to help the user achieve goals within the game with maximum effectiveness. Some of the interaction techniques used for non-game VEs are found in games as well, and have been inspired by them: basic navigation techniques, object collision, use of maps, choices of viewpoints, and awareness of other players.

Other 3D graphical environments are 3D MUDs (Multi User Dungeons), and 3D teleconferencing environments. Both are also multi-user applications (see section 2.2.4), and MUDs typically incorporate simplistic kinds of interface agents.

A MUD (Multi-user Dungeon) is a system where people create characters that can walk around rooms in a virtual world and interact with each other. There are some MUDs that are graphical and 3D. A commercial mainstream 3D MUD is Active Worlds. Active Worlds combines a 3D environment with a Web browser, which enables the combination of Web pages with locations in the virtual environment. Various projects have been built within the Active Worlds environment, including art exhibitions, games, education environments, shopping malls, etc.

A 3D teleconferencing environment is in some respects similar to a MUD, though with a more specific purpose. For example, the MASSIVE (Purbrick and Greenhalgh, 2000) system is a testbed for creating such teleconferencing environments. Several applications have been built with it. [example]

Some like to see the 3D 'immersive' style of graphical interaction as found in VEs as an extension or successor of both GUIs and Web-based interaction. There have been various projects in this direction, such as the 3D window manager (Topol, 2000)

and the Web standard VRML (Virtual Reality Modelling Language) (Carey and Bell, 1997). An interesting analogy of VEs with Web-based interaction is that both are concerned with navigation. There have been several projects that draw upon this analogy (Munzner and Burchard, 1995; Anonymous, 2001).

Some work has been done on interaction guidelines specific for graphical interaction in VEs (Eastgate, 2001; Gabbard and Hix, 1997), though some have noted that VE-specific usability issues remain underemphasised in general (Eastgate, 2001). VEs often differ in this respect from GUIs in the sense that they incorporate immersion into the environment, navigation through a complex space, and interaction with objects in the 3D space.

Realistic 3D first-person navigation inherently means that not everything is visible or easily reachable at all times. This introduces navigation and orientation issues, which are of a different nature than visibility issues in GUIs. Users might get lost or even get stuck navigating around tricky areas. Users require sufficient navigation facilities, such as proper orientation cues, a well-structured environment, landmarks or signs, and alternative types of navigation, such as a map versus location names versus first-person perspective.

Interactive objects in a 3D space are more difficult to interact with than GUI objects. Besides being obscured, objects may also be too small to see, or out of the field of vision. Interaction may be facilitated by making it easy to get a good point of view, provide clear selection feedback, enlarging relevant objects, use of transparancy to reveal obscured objects, etc.

On the technical side, the most often-recurring issue is the modelling of the 3D graphics and animations. Next to that, there is collision handling, and human kinematics. Collision handling needs to be done efficiently, and in particular collision with walls and walking over non-flat surfaces are difficult to handle correctly. Kinematics concerns the movement of limbs of humanoid characters. Usually, you want, for example, an arm to move into a natural way when the hand should be in a particular position to grab something. Determining this automatically according to some model of limb positioning and movement is called *inverse kinematics*. The paper (Vosinakis and Panayiotopoulos, 2001), describing the SimHuman system, gives a nice overview of some of the issues.

2.2.2 Interface agents

An *interface agent* refers to the interaction style of presenting a user interface in the form of an anthropomorphic conversation character. This style is typically meant to increase naturalness of interaction. We may distinguish two properties of interface agents that serve this purpose. The first is the *conversational style* which indicates that the agent and user interact as if it were a human-human conversation. The second is *natural language*, which is the ability of the agent to understand some subset of natural language.

The conversational style forms an interesting complement to graphical interaction. Some modern software introduces the concept of ‘wizards’, which is a conversational

interface style that is offered as an alternative or complement to graphical styles. Such conversational interfaces guide the user through a process step by step. A detailed comparison between conversational and direct-manipulation styles is given in (Stein and Maier, 1995).

The term *interface agent* is also used in the theory of *software agents*. Various literature on this subject has appeared which discusses the concept of software agent, and its many applications. The term *agent* is often used as a metaphor to aid in thinking, and need not be related to any formal properties (Wooldridge, 1999).

Various different definitions and classifications of agent systems exist (Jonker and Treur, 1998b; Watt, 1997) (Veer et al., 1998, the lecture by P. Braspenning). The classifications typically define agents in terms of metaphors, such as intelligent, intentional, autonomous, pro-active, learning, cooperating, and social. Some classifications also include the general application areas in which software with such properties may be used, such as information agents, internet agents, and interface agents.

Typically, an agent is defined as a system having, roughly, some of the following properties:

- 'autonomy' or 'pro-activeness': it is able to initiate relatively complex actions, or initiate actions without needing a specific trigger or command, for example after waiting a certain amount of time.
- 'intentionality': it is able to reason, and have explicit or complex intentions.
- 'communicativeness': it is able to communicate complex semantic and pragmatic information or is capable of negotiation with other agents.

So, the term *agents* refers to several things that have a certain amount of overlap. Agents are not only a user interface style (interface agents), but also a software engineering concept (see chapter 4), and a human-computer system modelling concept (autonomous agents: agents that do specific tasks autonomously). Usually, autonomous agents provide a model for delegating tasks to the computer. Furthermore, the roles of humans and software modules may be made more symmetrical by modelling them as agents which are assigned tasks (Palanque and Bastide, 1996). Some argue that autonomous agents can most effectively communicate with users if they are also interface agents (Watt, 1997), because this would allow integration in human environments. Autonomous agents introduce new issues. The article (Norman, 1994) discusses social and ergonomic aspects of humans using autonomous computer agents. Some guidelines given are: keep a feeling of control, and make sure people don't overexpect regarding agent's intelligence. Safety and privacy issues should also be considered. Agents also have some overlap with simulation models of human cognition, which we will discuss in chapter 4.

A prime example of an interface agent based UI is the *natural language dialogue system*. This is a system that speaks and understands human language (or natural language, NL) and is able to hold full conversations with a human. When dialogue systems first became feasible, one was rather optimistic about the possibilities. They received particular attention in the area of AI, as making a computer speak like a human was con-

sidered the ultimate in AI, as exemplified by the Turing test. Now, the state of the art is still limited. Natural language processing (NLP) is said to be an 'AI complete' problem: one cannot understand NL unless one understands a lot about the world. The main application area of dialogue systems is situations where (1) the communication channel is limited to audio or text, or (2) the application is only used by incidental users who do not have time to learn a new interface.

The best-functioning dialogue systems are very domain specific: they can only understand enough to perform the most necessary conversations within a given domain. Typically, dialogue systems are hardly built on advanced AI techniques, but rather, they are designed by finding out what the most often occurring dialogue patterns are, and supporting only these. We will now have a look at some dialogue systems that use some of the most advanced techniques.

The most common application domain is that of enquiry: giving search instructions for searching a database, and possibly doing simple modifications in it (such as reservation or ordering of an item). The dialogue system is in fact a search engine. A few examples of such systems are Dialogos (Albesano et al., 1997) (a railway enquiry system), ARTIMIS (Sadek et al., 1997; Bretier and Sadek, 1997) (a service enquiry system), and PADIS (Bouwman, 1998) (Philips Automatic Directory Information System, a phone directory service). These three systems operate through the telephone.

Less common are agents which are capable of some level of cooperation: they can help perform well-defined tasks. These may be considered more difficult to build than enquiry systems. A few examples are COSMA (Busemann et al., 1997) (a natural language system for appointment scheduling that works through email, processing available times, and making proposals), TRAINS93 (Sikorski and Allen, 1996; Stent and Allen, 1997) (a system for route planning of cargo trains), and the circuit fix-it shop (Smith and Hipp, 1994) (a system for diagnosing and repairing electronic circuits). In the circuit fix-it shop, the user is capable of doing the physical actions to repair the circuit, while the computer knows what to do. The computer walks the user through a plan to diagnose and repair the circuit, and adapts to the information given by the user. Here is a sample dialogue:

System:	This is the circuit fix it shop. How may I help you?
User:	RS111 is not working.
System:	I am familiar with that circuit. The LED is supposed to be displaying alternately flashing one and seven.
User:	The LED is off.
System:	The power is on when the switch is up.
User:	The switch is up.
System:	The switch is connecting to the battery when there is a wire between connector one one one and connector one two zero.
User:	LED is displaying one and seven.
System:	Is the one on the LED flashing?
User:	No.
System:	Is the seven on the LED flashing?
User:	No.
System:	There is supposed to be a wire between connector three zero and connector seven zero.
User:	RS111 is working.
System:	Glad to have been of assistance. Good-bye.

Note that the user re-wires the circuit without telling the computer, but the computer is flexible enough to adapt to the new situation. While this application domain is still limited, this dialogue indicates that this kind of system has considerable potential.

In the following paragraphs, we will cover some of the aspects of interface agents in more detail. In particular, we will cover natural language and cooperation.

Natural language

NLP systems are difficult to build, and building a system that understands NL to anywhere near the level of a human is more or less considered impossible. One of the most common practical problems that surfaces is that of ambiguity: when parsing a NL utterance, one often finds that there remain several possible ways to interpret it, which can only be resolved by a deeper understanding, which is not always possible. Speech recognition is a separate and complex problem (Heeman and Allen, 1997), and speech recognition systems are often seriously limited in the words that they can distinguish. Many NLP systems assume the speaker uses typed text instead of speech for this reason.

There seems to be no really successful approach to NLP that works by processing NL to a deeper level. In practice, shallow understanding techniques are found to work as effectively as attempts at deeper understanding. For example, for the understanding of an individual NL utterance, scanning for keywords typically works as well as attempts to parse the sentence structure. NLP systems only work for limited tasks in a limited domain. The domain in which they have been shown to be particularly effective is information retrieval. In (Androutsopoulos et al., 1995, page 7), the NL interface is contrasted with non-NL conversational or command-line interface types in information retrieval systems. Some interesting advantages are given. It is better for some kinds

of queries, which would require lengthy notation in other languages. Also, particular to NL is the existence of the context of what has been previously said, which allows briefer queries.

One NL modelling approach that is used very often is the *feature structure*. This is a data structure that represents the meaning of a NL utterance or the informational context of a dialogue. The structure consists of domain-specific information fields that may or may not have been filled in. Feature structures can be merged (this is called *unification*), enabling fields or substructures of each structure that have not been filled in to be filled in by the information found in the other structures. The VMC dialogue agent, for example, uses a feature structure based approach.

Dialogue structure and initiative. A separate issue in NLP is that of the *dialogue structure*, which is the discourse structure of one participant, and to initiative and initiative shifts between participants. Initiative, or control, refers to who is taking control of the interaction. Typically distinguished are mixed-, user-, and system-initiative, which may also be identified in other interface styles. For example, question answering is a system-initiative style, and command-line is a user-initiative style. In NL dialogue, choosing initiative for particular situations is considered a part of design choices. For example, mixed versus system initiative in certain NL dialogues is examined experimentally in (Walker et al., 1997a).

Mixed initiative is the most difficult to model, as it requires prediction of possible initiatives at each point in the dialogue. An often-found approach to model dialogue structure is the dialogue grammar approach (Chu-Carroll and Brown, 1997; Iwadera et al., 1995). This amounts to classification of utterances into a limited number of classes (the dialogue grammar, distinguishing for example question and assertion classes), and use these to determine control shifts. Sometimes, the dialogue grammar approach is meant merely to give some extra cues to predict control shifts. Shifts in initiative are usually modelled either by means of a finite state automaton or a hierarchy of dialogues and subdialogues. (Walker, 1996) contrasts a linear with a hierarchical model of control shifts. Since old utterances are forgotten, it is argued that control shifts are never completely hierarchical. An alternative approach is a plan-based approach, in which the system keeps track of the user's intentions. In (Jönsson, 1993), this approach is compared to the plan-based approach. It is argued that, in practice, the dialogue grammar approach works as well as the plan-based approach.

Cooperation. Some theories address ways of tracking and aiding a person's intentions. There are various that assume that intelligent cooperative behaviour can be achieved by the achievement of a plan. Several varieties of plan-based models exist: one can model one joint plan, one plan per agent (Ramshaw, 1991; Kitano and Ess-Dykema, 1991), or multiple (tentative) plans for a single agent (Ramshaw, 1991). In the cooperation and tracking of intentions, one may distinguish dialogue cooperation from task cooperation. In systems where complex explanations are required, one may go a step further, and distinguish discourse planning from task planning (Moore and Paris, 1989; Lambert and Carberry, 1992). The generated discourse is typically

textual, but may also be multimodal or multimedia, such as found in the Cosmo system (see section 2.3).

Dialogue cooperation is well exemplified by the well known Gricean cooperative principles (Grice, 1975): quantity (make response exactly as informative as required), quality (do not lie, do not state something you are uncertain of), relation (be relevant), manner (avoid obscurity of expression, ambiguity, be brief and orderly, etc.). A related, more formal, theory is is missing-axiom-theory (Smith, 1996), which is a theory of human reasoning. In this theory, goals are modelled as the intention to prove certain propositions. Wanting to obtain missing information (missing axioms) needed to prove the propositions is what results in dialogue.

According to (Levinson, 1987), people understate and oversuppose. If anything turns out to be unclear, this will generally be detected later, and can be explained afterwards. For NLP systems, this means that good dialogue management may compensate for bad speech or language understanding (Fraser, 1995). A simple guideline that appears natural and works well is the paraphrasing of what was understood by the system, such as is illustrated by the example dialogues in chapter 2.

Some literature discusses non-shared plans in non-cooperative dialogues, where a minimum acceptable level of cooperation is achieved by means of discourse obligations (Jameson and Weis, 1995; Castelfranchi, 1991).

2.2.3 Multimodality

There have been various systems that classify themselves as 'multimodal' systems. Multimodality is the use of different communication channels, such as pointing, linguistic utterances, or facial expressions, in an intermixed way.

A classification of the ways in which combinations of modalities can be used to convey information is found in (Martin, 1997). He identifies equivalence (modalities can convey the same information), specialisation (a modality is used for a specific subset of the information), redundancy (information conveyed in modalities overlaps), complementarity (information from different modalities has to be integrated to arrive at coherent information), transfer (information from one modality is transferred to another), and concurrency (information from different modalities is not related, but merely speeds up interaction).

What is typically called a multimodal system is a system that mimicks and understands humans' natural use of multiple modalities, which typically consists of a backchannel that is used to augment NL utterances. Like NLP systems, such systems require some level of intelligence to operate, though multimodal systems are not necessarily more difficult than NLP systems. Multimodality often helps to decrease ambiguity by supplying additional context. The paper (Nagao and Takeuchi, 1994) gives a classification of the multimodal interfaces typically encountered:

1. NL and deixis (i.e. pointing to something), gestures, or direct manipulation,
2. NL and contextual feedback (such as the current location or an object one is holding),

3. NL combined with facial expression.

Multimodal systems are well adaptable for use in a VE, as the use of such communication channels is natural in a spatial environment, and is a natural enhancement to the concept of presence. Some systems produce multimodal output, while others interpret the user's multimodal input, which requires more advanced AI techniques. Such multimodal input systems have not yet entered into the mainstream. Various experimental systems exist that combine the speech and pointing modality in tasks, often involving a map. We give a few examples below. Multimodal output, such as NL combined with facial expression, is typically found in interface agents, which are often part of 'state of the art' VEs, which will be discussed in section 2.3.

A typical example of a system with multimodal input is InterLACE (Trafton et al., 1997; Wauchope, 1996), a system for planning battle tactics, combining pointing on a map with typed text. It includes a full natural language engine, which understands commands and questions, such as *head north on road 4 to Grimma* or *is Wurzen closer than Grimma?*. Deictic references are supported, by using the reference words 'this' and 'these'. The computer can show items on the map when given an appropriate command.

Similar examples are CARTOON (CARTography and cOOperatioN between modalities) (Martin, 1997) (which assists in obtaining information from a map) and QuickSet (Johnston et al., 1997) (which is another assistant for planning battle tactics). CARTOON integrates speech, typed text, and pointing with a mouse. The user can query the map using commands similar to InterLACE. In QuickSet, the user can supply speech combined with gestures on a map, which involves gesture recognition as well. The gestures recognised are out of a limited set of symbols, such as fortified line, area, tank platoon and mortar.

We defined a modality as a specific channel through which information can be conveyed. If we look at it more theoretically, multiple channels may be identified in many ways, each with its own distinct properties, and each, in some way, separate from the others. The first question that comes to mind is: what channels are there? The answer probably depends on your viewpoint: in what way are the channels meaningfully distinct and separate? We will not take one viewpoint here, but instead, show some of the options available.

If we look at human perception, the most 'objective' answer is probably that a modality corresponds to a human sensory and motoric subsystem, like the eyes, ears, hands, and voice. When we consider a 'deeper' cognitive level, we might also say that, for example, people think differently about written language than they do about graphs, even though both are visual, suggesting another classification of modalities (Reiner, 1995).

On the other side, if we are considering technological issues, we might view modalities as corresponding to the computer's input and output subsystems, like screen, speaker, keyboard, mouse, and microphone. These may be considered distinct modalities because they require different manners of operation by the user. A classification according to a 'deeper' level of technological issues could also be made, for example,

continuous speech versus isolated-word speech, or screen versus paper.

Bernsen (Bernsen, 1996; Bernsen et al., 1998) presents *modality theory*, which classifies modalities according to both these cognitive and technological viewpoints. for example, written text, typed text, typed keywords, continuous speech, and isolated-word speech are different modalities. Properties of each modality are given, so that the effects of a certain choice of modality or modalities can be predicted. The emphasis of the article lies on the speech modalities, and on when to choose for or against using speech input or output. Another article contrasting advantages and disadvantages of voice to other modalities is (Cohen and Oviatt, 1995). This detailed model of modalities is useful for making choices of modalities in just about any kind of user interface. In fact, any user interface may be considered multimodal in this manner.

Various research has been done to observe human behaviour with respect to multimodality. Particularly complex is deixis. Deixis may be conveyed through looking at something, or turning one's body towards something, or by pointing or other gestures (Lester et al., 2000). The meaning of a deictic reference may depend on its precise timing, for example, it may occur at the precise moment that an anaphoric word or expression occurs (such as the user saying 'this') (Martin, 1997).

Technically, multimodal information may be incorporated in NL understanding in a straightforward way, for example by means of unifying additional feature structures, which is the approach used in QuickSet (Johnston et al., 1997).

Additionally, when multimodality concerns reference to the (virtual) environment, such as found in the case of contextual feedback, the system needs to be able to query information about the objects in the environment. For this reason, some VEs enable some semantic information about the virtual environment objects to be queried easily (Robert et al., 1998; Doyle and Hayes-Roth, 1997).

2.2.4 Multiple users

Various software exists that can be used by several users at once, so that they can effectively perform tightly coupled collaborative work. Existing collaborative tools are typically GUI applications with added multi-user features, often combined with chat or teleconferencing features. The chat or teleconferencing does not have to be an inherent part of the software, may be done through separate means. Collaborative applications may be classified into (Cosquer et al., 2000): shared drawing tools (shared whiteboard), cooperative (text) editing, group decision support (brainstorming or structured discussion), concurrent engineering (such as software development, but also other kinds of engineering). Collaborative software may be considered quite complex, as it combines several complex issues, in particular GUI, distributed computing, and multi-user issues. It is still far from common, but some interesting possibilities may exist here.

As mentioned before, MUDs too are inherently multi-user applications. The traditional MUD has a text-based interface: events are described in text, and the users communicate by typing text and use textual commands to do actions. MUDs are originally role-playing games, but they often have a social function too, and there are also special variants which are not games but are for social activities only. They provide various ob-

jects, rooms, and computer agents, to support or enhance various social activities. The users' avatars can typically be customised to a great extent, giving the users a great deal of expressive embodiment. In most MUDs, users can create their own objects and rooms using specialised programming facilities. MUDs typically have all of the properties of a VE, except that most are not graphical. While MUDs are mostly for entertainment and social activities, more serious applications using similar systems could be imagined. While computer agents are a common part of MUDs, they are generally unintelligent and/or they are not built for any purpose other than entertainment.

In the literature, multi-user software is typically called *groupware*. The study of groupware is called *Computer Supported Cooperative Work (CSCW)*. A common classification of different kinds of groupware is the context of use, that is, whether the different users are simultaneously present (same time or different time) or are in the same location (same place or different place). VEs typically fall into the category same time, different place.

What functionality should groupware provide? We may view some groupware as regular interactive software plus group functionality. Other groupware does not have any other purpose than group communication. A model for requirements on groupware of the first kind is the Clover model (Calvary et al., 1997; Coutaz, 1997). It distinguishes three aspects:

- **Functionality.** The regular tasks which are not related to the group.

- **Coordination.** This includes the regulation of concurrent tasks, such as floor control (deciding who has the turn), etiquette, and other social protocols. Typically, social protocols automatically emerge during the usage of a system, and the system should facilitate these. Another aspect of coordination is visibility: being able to see what others are up to. One principle that has been proposed is WISIWYS (What I See Is What You See). One example is the use of telepointers, which are the mouse pointers of other users that are visible on each user's screen. In virtual environments, some level of WISIWYS-like awareness may be created by the use of avatars.

- **Communication.** Users almost always need to communicate directly to achieve some task, so communication facilities should be present.

Technical aspects of groupware are mostly those of distributed systems. Especially troublesome is the latency problem. In practice, the network latency between users is so large that communication delays may be annoying and decrease usability. This problem may in part be solved by special design approaches, such as trading off view consistency for speed (Bhola et al., 1998). In graphical VEs, dealing with latency in a general way may be very complex (Ryan and Sharkey, 1998).

2.3 State of the art VEs

In this section, we will discuss the state-of-the-art VEs. These are meant to explore new user interface styles, such as advanced VE interaction techniques or interface agents. The systems we will discuss here have almost all of the identified VE properties. Figure 2.1 shows screen grabs of some of the systems.

- **Adele** (Shaw et al., 1999) is a medical tutoring system that runs in a Web environment. The tutoring is achieved mainly through simulation of medical cases. It combines Web pages, a GUI, and the interface agent Adele, after which the system is named. The Adele agent monitors the student's progress, and gives hints, answers questions, and explains. The agent is capable of pointing and facial expressions. In this system, the user cannot use NL or multimodality, and the user interacts with the GUI and the agent through more traditional GUI interaction strategies. Adele is part of the ADE (Advanced Distance Education) project of the CARTE group at the university of Southern California.

- **Steve** (Rickel and Johnson, 2000) is a virtual reality tutoring system for machinery operation. The user can perform tasks in a simulated 3D environment, and an interface agent (called Steve) instructs the user by conversation and demonstration. The user can use speech to communicate. Steve originates from the same research group as Adele, and they are said to share some of their underlying architecture. An important difference with the Adele system is that Steve is embodied *within* the simulation environment, and can participate in performing tasks. This way, Steve can demonstrate tasks, and can also play the role of a collaborator in a multi-person collaborative setting. Steve illustrates a broad range of possibilities that such an embodied situated agent has. Steve understands some NL in the form of speech, and a limited form of multimodality is supported by having Steve track the user's actions. Steve, and its predecessor TOTS, are part of the VET (Virtual Environments for Training) project of the CARTE group. Other VET agents currently under development are PAT and PACO.

- **Cosmo** (Lester et al., 2000) is a tutoring system for teaching internet protocol basics. Cosmo is the name of the tutoring agent, which is situated in a 3D world. The agent is able to explain the workings of some internet protocols, and is aided by graphical representations. The user is given tasks to perform or questions to answer. The agent uses facial and bodily expressions in conversation, and may point to particular objects in the environment. This system again emphasises output rather than input, as the the user interacts with the system through more traditional GUI-like interaction. Cosmo, and several related projects, such as CPU city, WhizLow, and Herman the Bug, are projects of the IntelliMedia group at the North Carolina state university.

- **AutoTutor** (Graesser et al., 2001) is a tutoring system consisting of a dialogue agent with an animated face, combined with graphical representations. It has been applied in computer literacy and conceptual physics courses. The main interaction with the system is through a NL dialogue, which the tutor combines

with facial expressions and gestural references to the graphical representations. The user communicates by means of typed text only. The NLP capabilities of the system are relatively advanced, and the emphasis of this system is on maintaining a tutoring dialogue, with little in the way of a supporting environment.

AutoTutor, and the related project HURAA (Human Use Regulatory Affairs Advisor) that is currently in early development, are projects of the TRG (Tutoring Research Group) at the University of Memphis.

- **Rea (Real Estate Agent)** (Cassell, 2001) is a system for selling houses. It combines a 3D environment which is a representation of a particular house with a conversational agent embodied in the environment. The environment is presented by a screen, while the user's actions are tracked by means of microphone and camera. Both the user and agent are able to use both speech and gestures to communicate. It does not use GUI interaction strategies, but simulates 'natural' human-human interaction.

  Rea, and its predecessor Gandalf, are part of the conversational humanoid project at MIT, based on the Functions, Modalities, Timing, Behaviours framework (FMTB, sometimes called FMBT).

- **SmartKom** (Wahlster et al., 2001) is a general information service system that combines graphical representation with an embodied interface agent. The user can communicate by means of a touch screen, speech, and facial expression. One can also choose to use speech alone to operate the system. The aim of the project is to provide information in 'non-desktop' environments, for example, an information kiosk or a handheld computer. The system aims at providing 'natural' human interaction, but also combines this with graphical representations. It even supports gesture recognition. A novelty is the addition of facial expression recognition, which is used to help disambiguate the user's utterances.

  The SmartKom project is a continuation of the AiA (Adaptive Communication Assistant for Effective Infobahn Access) (André et al., 1997) and PPP (Personalised Plan-based Presenter) (André et al., 2000) projects, originally developed at the German DFKI.

- **VMC (Virtual Music Centre)** (Nijholt et al., 1998; Nijholt and Hondorp, 2000) is an inquiry and booking system for a cultural theatre called the Music Centre. It consists of a dialogue agent for inquiry and booking of theatre performances, situated in a 3D VE, which is a representation of the music centre. Within the environment are various references to events at the music centre that can be examined. The system is Web-based, and combines a VE, a GUI, and Web pages. There is also a multi-user version. The dialogue agent uses some facial expressions and deictic references to indicate dialogue turntaking and direct attention to graphical elements. The agent is, as yet, not aware of the user's actions within the VE, and does not support multimodal user input.

  The VMC is a project of our Parlevink group at the University of Twente. The VMC dialogue agent is a continuation of the Schisma dialogue project. A related project is the UWISH navigation assistant project (van Dijk et al., 2001).

- **Jakob** (Evers and Nijholt, 2000) is a tutoring system that teaches a concrete task to be performed in a 3D VE. Similar to Steve, the user can perform tasks through interaction with the VE, and Jakob supports this by demonstrating, commenting, and explaining. The current version is in its prototype stages, and is being tested with the towers of Hanoi task.

  Jakob is a project of our Parlevink group at the University of Twente. Another ongoing Parlevink project in the same vein is the virtual piano teacher (Broersen and Nijholt, 2002).

There are two main applications areas that these systems are used for: education and information service. While these systems show some of the potential of VEs combined with interface agents, as yet such systems have not really entered into the mainstream.

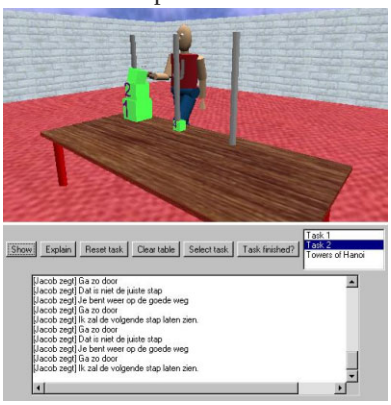
2.4 Conclusions

In this chapter, we discussed the four properties we identified in chapter 1 as characteristic of VEs, and gave example applications and some theory of each. This will be used as a basis for further discussion of topics on the different characteristics. Of the four properties mentioned, multimodality and interface agents are the least well established, and require some AI to be implemented effectively. But, as we will argue, it is also the combination of the different properties that makes VEs more difficult to model, as it requires an integrated framework for modelling all of them.

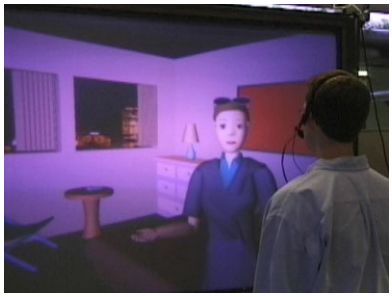
We summarise the properties in figure 2.2, and an overview of all applications we mentioned in figure 2.3. We split three of the four properties into further sub-properties, according to our observations in section 2.2, in order to refine the classification in figure 2.3. In graphical interaction, 3D interaction and animation is identified as a separate property. Multimodality is split into pointing- / embodiment-type multimodality, and facial expressions. Interface agents are split into NL understanding agents and cooperative agents.

Alphaworlds

Cosmo

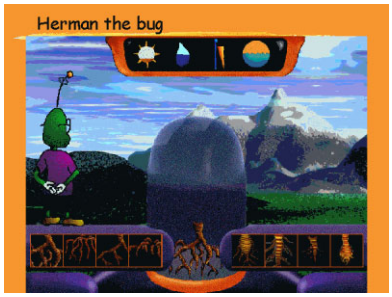


Jakob



Rea

Adele



Herman the Bug

Figure 2.1: Screen grabs of some VEs

graphics: Graphical interaction
Many VEs are very graphical in nature, and have the user interact with graphical elements. GUIs already incorporate a variety of graphical interaction patterns. For example, there are multiple windows that can be dragged and resized separately. Also, there is dragging of icons, and interactive selection from menus and lists. Other VEs typically incorporate interaction with life-like objects, such as doors, elevators, drawers, and bags.

- **inter**: Graphical interaction, such as dragging or operation of graphical representations of other physical mechanisms.
- **animat**: Depiction of environment with a great degree of realism, containing complex animation, or represented in 3D.

multimodal: Multimodality
We have already briefly mentioned the concept of multimodality. Multimodality is communication through multiple complementary communication channels such as speech, pointing, and facial expressions. It is a natural part of human communication. Multimodality may be used in a human-computer system to improve efficiency of information transfer, enabling the most suitable modality to be used for each type of information. It may also be used to improve naturalness, enabling users to communicate in a more 'naive' way.

- **point**: 'Smart' integration of pointing or actions of avatar with verbal interaction
- **exprss**: Use of bodily and facial expressions to augment or emphasise (verbal) communication

agents: Interface agents
An interface agent is a user interface style where the computer presents itself to the user as a conversational partner. The user performs tasks through a dialogue with the agent. While interface agents are not essential to the concept of VE, they are often found in VEs, where they usually have some kind of embodiment. Interface agents are especially suitable for giving information or advice, and may be used for exploration and learning.

- **natlng**: Agent uses free-form natural language to communicate
- **cooper**: Agent tries to act cooperatively

multi-user: Multiple users
VEs are often a medium that is used for remote communication. Such VEs are also called *distributed* VEs (DVEs), because such systems are necessarily distributed software systems.

Figure 2.2: Properties of VEs
The property names and abbreviations enclosed in boxes refer to the names as they are displayed in the table in figure 2.3.

System, name and reference	graphics		multimodal		agents		multi user
	anim at	inter	point	expr ss	nat lng	coop er	
GUI application		X					
Web-based application		X					x
Conferencing tool		X	U				X
Text MUDs			u	U	x		X
Graphical MUDs and teleconferencing	X	X	u	U	x		X
Computer game	X	X	X			x	X
Multimodal system		X	I		X	x	
Dialogue system					X	X	
Adele		X	O	O		X	
Steve	X	X	iO	O	X	X	X
Cosmo	X	X	O	O		X	
AutoTutor	X		O	O	X	X	
Rea	X		IO	O	X	X	
SmartKom	X	X	IO	IO	X	X	
VMC	X	X	iO	O	X	X	X
Jakob	X	X	iO			X	

Figure 2.3: Classification of a number of different applications

The letters in the checkboxes have the following meaning:

X = this type of system typically has this property.

For multimodality, there are several ones:

I = computer can interpret user's multimodality
O = computer presents information in a multimodal way
U = users can use multimodality to communicate with each other
X = I, O, and U

The letters may also occur as non-capitalised letters:

x,i,o,u = this type of system occasionally has this property, or has it in a limited form.

Chapter 3

The development process: a framework

3.1 Development: the evolution of a human-computer system

System development is the evolution of a human-computer system as it is being worked on. The issues of system development include those of software systems in general, which have been studied within the discipline of *software engineering* (SE). Software development is considered a hard problem, some even speak of an ongoing 'software crisis'. In human-computer systems, the tight interaction of the system with users forms an additional complex element in system development.

This chapter is meant to set the general context for the assessment of specification languages. It is an overview of 'best practices' used in development of interactive systems and VEs. We will pay particular attention to what is actually done rather than what is often prescribed. While one finds many SE-oriented overviews of development, these usually do not include other practices, such as used in development of GUIs, dialogue systems, or multimodal systems. Here, we try to combine the different practices into a general framework. From this information, we arrive at a set of requirements for specification techniques, which we discuss in chapter 5.

It is well known that development is an evolutionary process: it is not like having a clearly-specified set of requirements and writing down the solution. Usually, the requirements, goals, or constraints of the problem, nor the form of the solution are well-understood. Rather, they take shape as the system evolves. This is because the problem of creating a system is a complex one that cannot really be understood at a sufficient level of detail to see all the relevant issues. Rather, one can only see a small part of the issues at a time, and with this knowledge one can only work towards a better system in small steps, using the previous system as a basis for the next step.

Because the system requirements are not well-defined, and documented requirements are incomplete and change over time, system development means tracking a moving target. This means that it is not efficient to write a full system before one is more certain about requirements. At the same time, an existing system usually gives essential feedback regarding the validity and completeness of the requirements or goal statement. Effectively, a system and its requirements co-evolve.

Another problem of development is that of communication. Typically, there are multiple developers in a project. This is usually necessary to divide work and cover the different areas of expertise that are required. These parties will need to communicate. In some cases, communication is very tight, because much information from the different parties need to be combined to make proper decisions. It is not always possible to organise the development in such a way that concerns are well-separated between the different developers. As developers, one might have software engineers, experts on human-computer interaction, and domain experts (i.e. experts in the field for which the software is to be written). In some cases, some users actively participate in the development process as domain experts. Making users participate, having them give their opinion, and so on, is called *user-centred design* (Preece et al., 1994).

In our account of system development, we will look at the activities of development in a relatively detailed manner. Usually, system development theories describe development as a succession of *development phases*. In our account of system development, we will further describe these as *design-analysis cycles*, with incremental changes made during each loop of each cycle, until the phase is over. In fact, if we look at typical development activities in detail, very short cycles occur during most phases, in as far as phases can be clearly distinguished. For example, a programmer writes code by looking at the code just written, then writing or changing some, etc. We may say this is a design-analysis cycle, and, in fact, quite an important one (Green and Petre, 1992). After about 15 minutes (s)he might decide to try and compile it, then use the computational or formal feedback (compiler errors, assertion check failures, and behaviour) to modify the program some more. This is another design-analysis cycle. At the end of the day there may be a more 'formal' testing phase, testing the system with formal test cases and on different platforms. At the end of the week, it might be time to show the product to a select group of users, who might give some feedback concerning its usefulness and usability. We have described four development cycles of different length here, each contained within the other. The activities of the first two are typically lumped together as 'implementation', the third is called 'testing', and the fourth 'user evaluation'. We argue here that the precise activities of analysis, coding, documentation, and the feedback that is used in each process, are better understood with help of such a detailed account of the development process. This enables us to make a clearer coupling between theory on methodology and the detailed development process that we will be concerned with when we are considering specification languages. We can better answer questions such as: what is the use of a formal verification tool? Or: what specifications are needed in what phase? Or: what form should specifications take to be easy to understand or modify?

3.2 Controlling the development process

Before we start with a summary of existing development practices, we will introduce some general terminology. This describes the different aspects of a system under development. We will use the following basic notions:

- **setting (or setup)**: an environment consisting of humans and computers that work together in an organised way to work towards a specific goal. This includes interaction devices.
- **users**: the people involved in a setting, i.e. the users of the computer system under development.
- **(software) system**: the computer software involved in a setting.
- **task**: an activity that a user or users wish to perform, with or without help of a computer system.
- **mental model**: the model that a user has of the system. In the literature, it is also known as *conceptual model* (Preece et al., 1994).

We consider the choice of interaction devices (which are rather well represented in VEs and deserve mention here) part of the setting, rather than the system. This is because our concept of system, the focus of our research, mainly concerns the software.

While we consider settings with a tight interaction between humans and computers, much of the general terminology is the same for software systems in general. Usually, one distinguishes several types of activities that are part of the development process (Preece et al., 1994; Pasquini et al., 1998). We split development into to two basic activities:

- **Design**, which is describing some aspects of a (new) system.
- **Analysis**, which is finding out things about an existing situation, which includes people, software systems, tasks, and environment. We may distinguish two different purposes of analysis (Gibbon et al., 1997):
  - **Descriptive analysis**, which is to obtaining some kind of specification of what is going on, for example in the form of an explicit task, user, or system model.
  - **Evaluation**, which is analysis with the purpose of determining how well a (new or old) system fits its purpose. This usually means matching how the system would work in its environment against certain criteria.

We may view the activities of development as the design and analysis of a sequence of setups. Each setup bears some relation to the envisioned target setting, and is specifically created or chosen to gain insight that is needed as input for further development. In design, we typically specify some setup. In analysis, we observe this setup and draw conclusions. If we look at the information flow in such a strict manner, it is apparent

that many development activities consist of very short cycles. We view the system, the users(s), and the environment that determines the tasks and activities, as *roles* within each development setup. The notion of *role* emphasises that each of them may be filled in by a variety of role fillers. For example, the user role need not be filled in by a user, but is in some development techniques filled in by a computer system.

In software engineering methodologies in general, one usually finds various setups which only focus on the computer part. While we will emphasise those setups in which there is a human part, computer-focussed setups are still part of the development of human-computer systems.

The concept of human-computer setup and its various types of input and output are illustrated in figure 3.1. The figure is a summary of the different development setups that we will discuss in the next sections. Some of the various types of input into and output from this setup used in existing techniques are specified in the figure, representing different choices of user, system, and interaction, and different means to obtain information about them. Note that in some cases, the design part is insignificant, while in other cases, the analysis part is small or implicit. We may well describe many techniques found in development as just specific combinations of such inputs and outputs. We will explain this in following sections. In fact, some development activities that are not typically viewed as being centred round a setup, may rightfully be described as such, making some decisions more explicit. Next to that, various input-output combinations exist that are largely meaningless and will never occur in practice, such as having a human in the system role while having a computer in the user role. Next to that however, various input-output combinations may also be formed that are not used in existing techniques, but may be quite meaningful.

3.2.1 Tools to aid the developers

With help of this view of development, we can identify several distinct aspects to development:

- **design knowledge: design principles and guidelines**. There is much theoretical work describing what makes a good system and why. The knowledge may consist of anything in-between general theories, such as psychological or software engineering models, and libraries of specific solutions for specific problems.

  There has been some work on how to properly record, retrieve, and make use of design knowledge. One approach that received particular attention is *design patterns*. A design pattern is design knowledge specified as a series of solutions of often-occurring problems, in a concrete problem→solution format. The solutions are specified as precisely as possible, using specifications that are closely mappable to the implementation language.

- **design tools: tools for enabling input into a setup.** Usually, with design we mean the set of *specifications* that determine the behaviour of a given setup, such as prototypes, program code, and task models. These form the most substantial

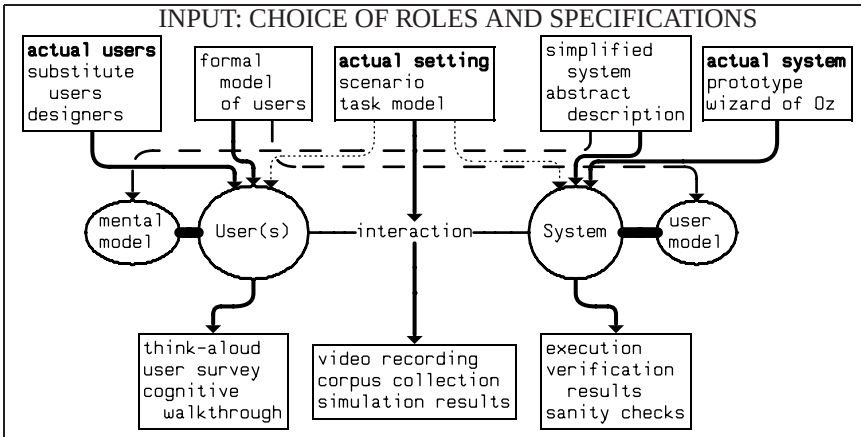


Figure 3.1: Possible system and user setups during development

This figure illustrates the different roles in a setup and summarises the possible setups by enumerating the range of possible inputs and outputs. The combinations of inputs and outputs that are used in actual development techniques are explained in the next sections.

In the middle are the user(s) and system, and their interaction, which is a result of some given tasks, goals, or environment. Note that in some setups, the users may actually be a computer simulation of users, and the computer system may be mimicked by a human.

At the top is the design input: the different choices of user, environment, and system for a setup. Note that the choice of environment or setting (drawn in the middle) is specified here to determine the interaction. Actually, the environment determines the tasks and goals, which in turn determine the interaction. The environment also includes the broader context of organisational goals or settings, but these will not be addressed here. In some setups, the environments are deliberately chosen to be artificial and limited, such as is the case with prototyping using scenarios.

At the bottom is the analysis output: the different kinds of observations that may be obtained from a setup. Note that the process of envisioning a setup by just perusing models and specifications is also a special kind of observation, in which the user, task, or system are just imaginary. An example where this manner of observation is formalised into an evaluation technique is the *cognitive walkthrough* method. Generally though, this envisioning is an informal yet ubiquitous activity that is usually considered part of the design activities.

The user and system may maintain explicit models of each other. These models are drawn as separate circles. Some of the models of the user and system that are specified may actually be used to shape the model that the system has of the user, and the user of the system, respectively. For example, system models may be fed into user simulations (see chapter 4 on user simulation models), or may be used as further material for interactive help systems (Palanque et al., 1993; Moriyon et al., 1994).

aspect of the input that determines a setup. Specifications may specify any aspect of a setup. Examples are specifications of requirements, user tasks, possible scenarios of use, prototypes, global system designs, or the code of a working system. There are a variety of specification languages, some of which are specially meant for specific aspects of a human-computer setup. A second aspect that is particularly relevant in some setups is the *experimental design*, such as the choice of users, instructions to give to users, and physical setting. The usual concept of experimental design, however, also includes analysis tools.

- **analysis tools: tools for obtaining output from a setup.** Obtaining output means *collection of data* (i.e. observing the setup), and *interpreting the data* (i.e. analysis). There are a variety of tools and methods for either. They eventually lead to detailed diagnostic information or further descriptive specifications.
- **methodologies: planning and sequencing of setups.** Basically, methodologies prescribe what setups should be made to gather the information required for further development, or what kind of planning should be done for future development activities.
- **development environments: software tools for interactive development.** Software development has long benefited from facilitating tools, such as debuggers that show program execution, and integrated editor/compiler environments. Other possibilities of software tool support are support for methodologies and semi-automated design.

These aspects will be introduced in more detail in the next sections.

3.3 Design

An introduction to design knowledge with respect to interaction was given in chapter 2. An overview of design models will be given in chapter 4. A detailed overview of design specification languages will be given in part II. What is left to describe in this section is the experimental design: the choices of role fillers that we have in determining a setup. We distinguish several types of role fillers for the user, system, and interaction roles:

The user role:

- **Real users**. In an analysis where the goals and tasks of the users, or the users' knowledge of and interaction with the system, are not well-known, the role of the user should be filled by actual end users, or possibly by other people representative of the end users.
- **Simulated users**. If enough general information about the users is available, one sometimes places a computer simulation in the user role. This is attractive in the sense that no experiments with users are needed, and effort and turnaround time is greatly reduced. The computer simulation model is typically fed a knowledge specification, and a task specification or a combination of a task and simplified system specification.

- **Developers**. In some cases, a developer (typically a HCI expert) takes the role of the user, trying to anticipate what a user might do in a given setup. These are the cases when either a given system or task are just being considered intuitively without a formal setting, or when an explicit psychological opinion from a psychology expert about the user's possible actions is desired.

The system role:

- **Full system**. A 'full system' is a version of the system as it should be, and forms the most complete role filler for the system role, but a full system is typically very expensive to develop. The system is defined by its program code.
- **Prototype or mock-up**. One uses a simplified system instead of a full system. This can be anything from a series of screen drawings to a more or less working system. Because of its low cost, prototyping is an important feedback tool in usability engineering (Nielsen, 1993). Prototyping also has disadvantages (Atwood et al., 1995): some assumptions about the system may be hidden in the simplified prototype, and sometimes the simplifications highlight inappropriate HCI issues. Additionally, there is often the tendency not to throw away the prototype but to evolve it into the full system, which has its disadvantages, because the prototype's primary purpose is to illustrate behaviour and appearance, and not provide a specification for the working system. Therefore, requirement specifications may form an appropriate complement to prototyping (Hall, 1997). Another solution is to ensure that the prototype is *both* appropriate as behaviour/appearance and system specification, which is a good idea only if this can be done with little extra cost.
- **Requirement specification** is design in the form of a specification of *constraints* that the system must conform to. These requirements may come in different forms. If they are precise, mathematical specifications, they can be analysed using mathematical proof techniques. Such specifications are also called *formal specifications*. If they are computer-readable as well, they may be checked automatically (i.e. *testing* and *verification*). Informal requirements, such as natural language specifications, may provide input for developers or users. Requirements capture an important part of the system, namely, the range of implementations that form a valid system. Requirements therefore capture extra information not even found in a full system. They may be specified at any level of detail, from overall system specified in little detail, to the input and output constraints of most parts of the full system. An important problem with requirements is that specifying requirements that are a *complete* coverage of what the system should do is often very difficult or impossible, neither is it very well possible to check *if* a given set of requirements is complete to any particular level. In practice, requirements cover just the most important or easiest-to-specify parts of the system.
- **Wizard of Oz (WOz) simulation**. One places a human in the role of the system. This is often done to obtain initial data for a system, when no working computer system is available yet. The WOz technique is known from dialogue systems

design (Sparck Jones and Galliers, 1996). WOz is only applicable to situations where the interaction is analogous to human-human interaction, and is relatively hard to achieve by the computer, but can be achieved more easily by a human. It is usually used for NLP systems, and sometimes for multimodal systems. A special case is the 'bionic wizard', which is a human wizard, heavily aided by computer tools to simulate the interface.

The environment role:

- **Real environment**. This refers to the real-life situation, which is also called a field study. It is particularly useful when little is known about environment factors, and the environment must be analysed first. In contrast, the task and environment may be designed carefully to reduce random factors from the environment when only users and system are concerned. This is called laboratory- or controlled study.
- **Scenarios**. A scenario is a sequence of steps that depicts a possible interaction of users with the system. Scenarios may be described in various levels of detail. They may be fruitfully combined with a prototype system. Using a scenario reduces the need for a full system and/or a real environment. Well known is the concept of *use cases* (Grieskamp and Lepper, 2000) which are scenarios described by a sequence of human-computer or computer-computer interactions.
- **Task model**. Task models are often made early in development just to gain some intuitive insight in existing setups. They are also used in user simulation methods, in which the task model determines what the user has to do.

3.4 Getting data for analysis

Prior to analysis, it is in some cases natural to identify a separate phase for obtaining data. There are techniques in which obtaining data is a distinct step prior to analysis. While data from the system is easily obtained, getting data from the interaction is slightly more difficult. Getting data regarding the user's mental state is particularly difficult.

Getting data from the setting or the human-computer interaction is typically done using some recording method, possibly followed by a transcription into textual format by hand. For example, in graphical user interfaces, one sometimes makes video recordings of the interactions, which are then transcribed into a text describing the events (Ek, 1997).

A less-used alternative to regular data collection is the determination of more abstract properties of the system, such as number of operations needed to do a task, visibility of relevant information, etc. (Mezzanotte and Paternó, 1996; Hussey et al., 2001). In this case, one only needs an abstract formal specification of the system, interaction, and user.

For the analysis of NL dialogue systems, one typically records the spoken or written text. A number of such recordings is also called a *corpus*. In the NLP literature, a large amount of work is done on using corpora for system development. In the proceedings (Andernach et al., 1995), using corpora for systems development is addressed. Using corpora for utterance type prediction is addressed in (Andernach, 1996) and (Keizer et al., 2002). Using corpora for obtaining a simulated-user model for evaluation is described in (Eckert et al., 1998). Sometimes a limited amount of interpretation is done on the corpus also. For example, the words are annotated with types and meanings. This is called called *corpus annotation*. (Dahlback et al., 1997): standardisation of corpus annotation for NLP systems.

There are several means of getting data about the user:

- **think-aloud method**. This is an attempt to gather immediate and detailed information about what the user is thinking when he/she is using the system, rather than information that has to be reconstructed afterwards. Note that the requirement to talk while using a system may interfere with the user's behaviour. This is obvious with speech systems, as the user is already using speech and language in his/her interaction with the system, but interference in the form of disturbance of cognitive activities is always present to some degree. See (Ericsson and Simon, 1980) for detailed information about issues concerning the gathering and use of think-aloud data.
- **user surveys or interviews**. Instead of using think-aloud, the user may also be asked about the system afterwards. For example, one may ask questions of the kind: *'why did you do that?'*, which can be used as an alternative to the think-aloud method. Other kinds of questions are evaluation questions, such as *'Did you find this feature useful?'*.

3.5 Descriptive analysis

Some types of analysis result in a descriptive model of an existing setup, which is obtained from collected data. We will call this *descriptive analysis* or *qualitative analysis*. Most often used are conceptual models (how the user sees the system) and task models (what the user wants to achieve and how) (Chase et al., 1993; Frascina and Steele, 1993). For example, the *Group Task Analysis (GTA)* method (Veer et al., 1996b; Veer et al., 1996a) prescribes how analysis methods, such as ethnography, should lead to descriptive models like Hierarchical Task Analysis (HTA) or *Methode Analytique de Description* (MAD) (both of which describe task models in terms of action hierarchies), and object models (which describe some informal properties of objects in the domain).

Note that there is a fine line between descriptive analysis models and design models. GTA for example prescribes the creation of two task models in the first development cycle, called 'task model 1' and 'task model 2'. Task model 1 is created by observing the existing setting. Task model 2 is the task model of the envisioned modified system. Both may be specified using the same specification language. In our terminology, task model 1 is a descriptive analysis model, and task model 2 is a design model.

3.6 Evaluation

Evaluation is the analysis of a system which results in information about the suitability of a system: whether it is good or bad. Evaluation requires raw data to be translated to information about the quality of the system. One may distinguish software evaluation and human factors evaluation. Software evaluation concerns the purely technical aspects, such as software quality measurement and fault detection. Human factors evaluation concerns issues such as task efficiency and user opinions. Note that a relatively large body of literature exists that covers evaluation of NL dialogue systems, which is in nature much like human factors evaluation, only more specific and often more rigorous (Sparck Jones and Galliers, 1996; Bernsen et al., 1998; Walker et al., 1997b). We will discuss some NLP evaluation techniques below.

Various general evaluation criteria for UIs have been proposed in the literature. Such general criteria are useful because they define a general scope of what to look at when evaluating human-computer systems. They are used to define more detailed evaluation techniques.

Well-known is the work by Jakob Nielsen (see for example (Nielsen, 1993)). Nielsen defines the criterium *usability* which is part of *usefulness* which is in turn part of a bigger system acceptability scheme, distinguishing 'social acceptability' and 'practical acceptability'. Usability is classified into: easy to learn, efficient to use, easy to remember, few errors, and subjectively pleasing. Note the distinction between learning and remembering: learning includes learning the general concept of the system, while remembering is only remembering the fussy details, while the general concept is already known. The criteria most often found in other literature are effectiveness, efficiency, satisfaction and learnability (Sparck Jones and Galliers, 1996; Ek, 1997; Haan et al., 1991).

There is reasonable agreement among the different classifications, save for the scope of the different criteria. When we take this to an extreme, any criterium may be seen as just an inessential indicator of some other broader criterium. In practice however, it is especially important that the criteria cover those things that are actual problems found in actual systems.

A remarkable and perhaps problematic feature of these classifications is that there is a subjective element contained in them: pleasant to use. While important, this is a difficult criterium, because it is hard to relate to system properties, and may be very individual. It may be a result of the other criteria: many users will find a system that they think does not do its job well unpleasant to use. Conversely, it may also be a result of factors not within the scope of the system or the control of the designer, such as social factors. For example, if social factors determine that users think that a specific type of system is not usable, it is likely that they also find it unpleasant to use.

We may distinguish several general types of evaluation:

- **Metrical evaluation.** Here, data from corpora or similar data logs are summarised by measuring and counting surface properties. A second source of data is user surveys. See (Minker, 1998) and (Hirschman and Thompson, 1996) for

general overviews of metrical evaluation. We may distinguish two steps in specifying the relevant aspects of quality (Sparck Jones and Galliers, 1996):

- defining **criteria**, which are (often abstract and general) descriptions of the relevant suitability aspects. Knowing what criteria you are interested in does not mean you can tell whether a certain system satisfies them. To do this, one may define:
- **measures**, which are readily implementable means to measure criteria. In the case of human factors, we may distinguish two kinds of criteria: *subjective* and *objective*. Subjective means that the data consists of people's opinions, as is obtained for example by survey questions such as the aforementioned *'Did you find this feature useful?'*. Objective means the data consists of measurable properties that do not contain such a subjective element, for example, the time taken to complete a task or algorithm. Note that obtaining extensive subjective information, like open interviews asking for opinions, actually leans towards user-centred development.

Evaluation by using metrics is practiced often in NLP evaluation methods. A lot of different NLP-related metrics have been proposed in the literature (Minker, 1998; Walker et al., 1997b; Danieli and Gerbino, 1995; Sikorski and Allen, 1996; Albesano et al., 1997; Polifroni et al., 1998; Dillon et al., 1993), with the most often-recurring being number of dialogue utterances, time taken, and number of errors made. Metrics are easy to use, but there is the risk that important information is not reflected in them. Metrics are often used to compare one system to another system or another version of the system (comparative analysis), or to check whether an existing system is acceptable. Comparative analysis is useful for verifying a redesign or for deciding whether to commit to the usage of a new system. More difficult is comparing multiple systems across multiple domains (benchmarking).

Metrical evaluation means summarising the quality of a system in abstract figures, which possibly means an oversimplification of the problem. For example, (Gilmore, 1995) suggests that GUI interaction is not always the best, even while it does yield the best response speed. In an experiment, he showed that a user interface with a slow response speed resulted in the fastest task completion.

Software engineering also uses *software metrics* (Kitchenham, 1996) to measure quality aspects of the software itself. The criteria measured include criteria relevant to the users, such as reliability and efficiency as well as those relevant to the developers, such as understandability and modifiability.

- **Specification walkthrough** is evaluation done by just reading specifications. It comes in several forms, and often, it is used informally during activities, and is not considered an evaluation method.

  A HCI-centred example is *cognitive walkthrough*, one uses a (possibly much simplified) specification of the user and system, instead of an actual experimental setup of user and system. Scenarios are used to specify possible interactions. The specifications are read by fellow developers, who discuss whether the scenario

will be followed by the user as specified, and whether the design is a good one for this scenario. Cognitive walkthrough is a usability evaluation method which neither requires users nor a full system, and is particularly useful in early stages of development.

Another kind of specification walkthrough that is suitable for HCI, but has a higher level of abstraction than cognitive walkthrough, is *QOC (questions, options, criteria)* technique, also known as *design rationale* (Jorgensen and Aboulafia, 1995). Here, design decisions are made by first formulating a design question, and then writing down a number of options and criteria, and judging which options do or do not satisfy each criterium well.

A software-engineering-centred example is *code inspection*. Here, code written by one developer is read and reviewed by several others in an open discussion. It is expensive if we compare this to regular programming activities, as the inspection procedure takes a lot of time, but to date, it is still one of the most effective methods of finding errors (Beck, 1999).

- **Testing or verification** is analysis of an implementation or other description of a computer system against some constraints, which may be either implicit and general constraints, or constraints specified in a separate specification. Essential to this approach is, that the constraints are formal in such a way that their compliance can be verified (semi-) automatically. While such constraints are not easily specified, this method does have important advantages. Firstly, once constraints are specified, evaluation is done for free. Secondly, with the method, one can reach a level of thoroughness that makes it very suitable for evaluation of software. Note that verification only points out errors and does not show how to fix them. This is not really a problem, since finding faults is generally more costly than fixing them (Green, 1990b).

  Verification is well used in SE. In fact, the very common and informal compile-and-fix activity usually part of the 'debugging' process already involves limited forms of verification. If a program compiles and it doesn't crash, we already have established limited but significant verification. Compilers have, over time, included more and more sanity checks. Beside syntax checks (which are already useful, as languages are often built in such a way that syntax errors point out underlying semantic errors) there is an increasing amount of sanity warnings, such as 'variable may be used before definition' and 'statement not reached'. There are other kinds of verification, such as assertion and model checking, which will be explained in more detail in chapter 5.

  Formal verification of usability properties would be an interesting feat, as even a slight reduction of user testing would mean a reduction of costs. Some work has been done in this area, but remains mostly experimental (Campos and Harrison, 1997; Lewin, 1997; McInnes et al., 1995). Typically, properties include reachability, visibility, reversibility of actions, and reliability (which means here: does the system do what the user expects?). An identified problem with most existing methods is the limited expressiveness of the allowed models.

Clear evidence showing which evaluation method is the best in what circumstances is

not easy to obtain. It is apparent though, that proper evaluation is beneficial even in simple systems. The case for software evaluation is easy to make, and various types of evaluation (metrical evaluation such as profilers, and verification such as assertion checks) are now considered part of basic software facilities. For HCI, the added value of evaluation is sometimes more difficult to prove, as the criteria are less formal. But, for example (Tullis, 1993) shows that evaluation is preferable over a naive choice of interface strategy. In their experiment concerning the design of a selection strategy, 'naive' designers turn out to be fond of the 'drag and drop' strategy, while evaluation points out that it is one of the slowest to use, nor is it the strategy that the users prefer. In other cases, the benefits of evaluation are not so clear-cut. In (John and Marks, 1997), for example, the effectiveness of some usability evaluation methods is compared, with doubtful results. Comparison is done by determining whether problems were identified, whether the problems led to a design change, and whether the design change was good or bad. The methods were tested on a partially-implemented multimedia authoring package. The results are not clear-cut, but it was concluded that they were generally unsatisfactory, especially since the 'simply reading the specification' method seems to come out relatively well.

3.7 Methodologies and development environments

We will define a *methodology* as a prescription of development activities. Methodologies may have various levels of detail and rigidity. Typically, a methodology distinguishes a number of phases or steps, each prescribing a specific kind of setup. Dependencies between the phases are specified, prescribing the information that has to be gathered as input for each phase. Most methodologies are closely linked with specification languages, as they prescribe which ones to use for input or output of each phase. Methodologies usually integrate a number of different languages.

Development environments are software environments that are meant to aid the developer through automation of development tasks. The most basic environments are specification tools, facilitating the specification process and enabling different specification languages to be integrated to some degree. Some environments go a step further, and contain heuristics to enable decisions to be made interactively. Such an 'semi-automated design' concept offers interesting possibilities, though it is less used.

Both development environments and methodologies provide ways to structure the development process, and provide an ensemble of specification languages (i.e. a specification technique) to do this. It is not surprising that some development environments are meant for use with specific methodologies. We may also distinguish a class of prescriptive models that provide just a specification technique without any well-defined method or supporting software tools. An example is *UML (Unified Modelling Language)* (Object Management Group, 2001), which also has an interactive variant *UMLi* (Pinheiro da Silva and Paton, 2000). The languages used in UML are well-established indeed, and exist in many variants (Schewe, 2000).

Conversely, there are various engineering-oriented methodologies which mainly de-

scribe process and not specification language or even the precise application area (Pasquini et al., 1998; Braude, 2001). Particularly well known are the *waterfall* model and the *spiral* model. There are many others. The terminology typically used deviates somewhat from what we have described here, which is a more detailed description of the information flow. The fact that each phase is in fact an iterative, cyclic process is not described explicitly in these models. As a general illustration, here are the phases in the waterfall and spiral model and their description in terms of our framework.

> **Analysis**: either a cycle or a single pass with as input existing users with an existing system, and as output some kind of requirements specification.
>
> **Design**: a cycle with as input a design document and as output intuitive feedback.
>
> **Implementation** a cycle with as input code and as output formal (or computational) and intuitive feedback.
>
> **Unit testing** a cycle with as input code and as output more extensive and thorough formal feedback. With each iteration, small on-the-fly fixes are made in response to errors found.
>
> **System testing** a cycle with as input the entire code and as output formal feedback. Like unit testing, small on-the-fly fixes are typically made.

Basically, the waterfall model has these phases occur in a strict sequence, and the spiral model iterates, starting from the beginning if the last phase is done, refining the system under development in several cycles.

Actual development often does not follow such strict prescriptions, and it seems these traditional methodologies have failed to some extent to achieve their goals (Kitchenham and Carn, 1990; Beck, 1999), as they fail to address some actual problems, in particular the problem of communication, and the inability to predict the future, even with extensive predictive models. For the case of HCI, Nielsen (Nielsen, 1995) also noted that, in traditional methodologies, communication between the different parties does not proceed easily, and that the methodologies imply considerable investment of resources before any return is obtained.

A class of alternative models have been proposed to address these problems, which are called *lightweight* or *agile* models (Highsmith and Cockburn, 2001). Examples of these are *extreme programming (XP)* (Beck, 1999) and *scrum* (Rising and Janoff, 2000). These models are different in a number of ways. In particular, they emphasise:

- minimal solutions and quick development cycles to account for the *inability to predict the future*. They try to adapt instead of predict.
- the importance of *quick feedback with users* or user-centred development, primarily using prototyping.
- the importance of the *program code* as the primary specification, as maintaining too much additional documentation is too heavyweight in short development cycles.

- design quality maintenance by means of making incremental changes that preserve functionality.
- communication, by means of code sharing practices, such as pair programming, ego-less programming, and regular well-defined meetings.
- thorough testing after incremental changes by means of automatic testing techniques.

Such models are particularly relevant for interactive systems, as a proper feedback cycle with users is one of the main issues of interactive systems development. It remains unclear whether agile methodologies are a significant improvement over more traditional methods. Arguably, they do have a point as concerning the identified issues. Some people argue for a combination of techniques from both traditional and agile methodologies (Turk et al., 2002; Kelter et al., 2002).

Since our main topic is specification, it is interesting to look at the design specifications prescribed in different development methodologies in a little more detail. Each specification has its own emphasis and purpose, and often, specific languages are prescribed for specifying each specification. We will give an overview of the different kinds of specification found in some existing methodologies found in different application areas in the table below. The areas which we will look at are general SE (Unified Modelling Language UML (Object Management Group, 2001)), general HCI (Method for Usability Engineering MUSE (Stork et al., 1995), the Lauesen method (Lauesen and Harning, 1993), UMLi (Pinheiro da Silva and Paton, 2000)), NL dialogue (the Bernsen method (Bernsen et al., 1998)), group-based HCI (Group Task Analysis GTA (Veer et al., 1996b)), Web and hypertext (Object Oriented Hypermedia Design Method OOHDM (Rossi et al., 1999)), and VEs (the CLEVR method (Eastgate, 2001)). The table lists a number of general topics of specification, which is chosen so as to cover most of the specifications used in each of the methodologies.

Description	SE	HCI			NL	cscw	web	VE
	UML	MUSE	Lauesen	UMLi	Bernsen	GTA	OOHDM	CLEVR
Structural models describing the objects in the domain of discourse and their relations, or the system data	X	X	X	X	X	X	X	
Sequence models describing events and system reactions, including the human-computer dialogue flow, or information flow between humans	X	X			X	X	X	
Task models describing user's tasks and possibly their decomposition		X	X			X		X
Screen layouts, graphical window layouts, or window structure		X	X	X				X
Requirements and design decisions models, specifying general properties of the system and early design choices in an abstract way		X			X			
Properties and purpose of the interactive objects			X					X
Dataflow models describing human or computer agents and their interaction	X							

While it would be premature to draw extensive conclusions from this small list, we might note that structural and sequencing models are the most common, and that, oddly, dataflow models are the least common.

Note that agile methods do not have the richness of specifications that traditional methods have, since they mainly work with the program code. To have the best of both worlds, specifications from traditional methods should be made part of the program code by integrating them into an executable model, or they should be generated automatically from the code (Kelter et al., 2002).

3.8 Conclusion

In this chapter, we introduced a framework for systems development, based on the idea of a design-analysis cycle, centred around representatives of the setting under development. We incorporated the set of best practices in this framework, and discussed specific alternatives used for developing systems which have the specific characteristics of VEs. Beside the fact that the terminology introduced here will recur in various places, this framework is used as a basis for specifying the requirements for specification languages, which is found in chapter 5.

Chapter 4

Design models

In chapter 2 we gave an introduction of one relevant part of design knowledge: design principles for interaction. In this chapter we will go into a bit more detail, and cover the relevant basic models that are used to think about and shape design. This includes models of cognition, and of the structure of software and particularly interactive software.

4.1 Models of the user

Cognitive science is the study of human intelligence. One of the most important models developed by cognitive science is the multi-store memory model (Veer and Lenting, 1995; Preece et al., 1994). This splits memory into sensory memory, short-term memory, and long-term memory. Especially the latter two are important for our research. Long-term memory is highly structured and has a high capacity, but it forms only very slowly over time, requiring repetition and training. Short-term memory is memory for storing arbitrary information but with very limited capacity. Oddly, short-term memory appears to store information as a number of discrete units, which are named *chunks*. It can hold only about 7 chunks at a time. Short-term memory stores information by referring to long-term memory, so that more information on a particular subject can be memorised when more knowledge about it is present in long-term memory. Well known are the studies of chess players (Gobet, 1999). Unlike novices, expert chess players are able to quickly memorise an entire chess board. This difference in capacity of short-term memory appears to be caused by the availability of more complex patterns of chess pieces as memory chunks.

Another relevant cognitive psychology model is the model of human error: typically distinguished are slips and mistakes (Pasquini et al., 1998). Mistakes are errors that are caused by misconception and misinterpretation, slips are accidental errors that always occur due to the imperfections of the human mind. It is interesting to note that slips almost always occur, even if the person is perfectly understanding and capable.

However, people have argued that traditional cognitive psychology models do not account well enough for context. Other frameworks try to remedy the situation (Nardi, 1992): there is *activity theory* (Kaptelinin et al., 1995), *distributed cognition theory* (Zhang and Norman, 1994), and *situated action modelling.* Basically, they assert that human cognition cannot be seen as separate from the environment, in particular, the tasks and tools in the environment. It is said that, as a human may shape a tool according to a task at hand, the tool will also shape the human, as the human is restricted to working with the limitations of the tool; humans and their tools co-evolve. More practically, these models call for an integrated approach to modelling the flow of information, as it considers knowledge as something that is distributed among people and artifacts. Representation and specification are relevant cases: an external representation is read (internalised) or formed (externalised). Thinking is seen as an internal activity with internal representations. Use of a specific external representation may make cognitive tasks easier or harder, and may eventually shape thought. Consider the following example: the two-player 'game of 15'. In this game, there are nine numbers, $1 \ldots 9$. Each player may in turn encircle a number. The player who has a sum of three numbers that equals 15 first, wins the game. As it turns out, this game is simply tic-tac-toe with a different representation. However, one may expect that the original spatial version is quicker to understand and play by most people than the numerical one.

4.1.1 Simulation models

There are a particularly large number of user simulation systems, which are summarised in this section. Usually, the simulation model is directly based on cognitive psychology models. In fact, we may say that they are just variants of these models, specified in enough detail to be executable. The user simulation model is usually fed a task specification, sometimes with a simplified system specification, which it then executes. Working with simulation models touches in this respect with task modelling techniques. In fact, the simulation models are often associated with specific task modelling languages, though simulation models and task languages may in some cases be separated and recombined. Another area which is related to user simulation models is modelling of cooperative interface agents. Some of these are fed user knowledge specifications which are similar to those used in user simulation models.

Various information may be obtained from such simulations. It is these models that are typically used in a user simulation setup. The models are usually used in combination with metrical evaluation. Additionally, such simulation models are sometimes used to model the behaviour of interface agents, as they are models of intelligent behaviour as well as behaviour that is analogous to human behaviour.

Most simulation models are based on task hierarchies. A task hierarchy consists of a goal which is composed of successively refined subtasks. The different systems use slightly different task models, emphasising different aspects of the user's activities in different levels of detail. Some task models enable some description of the objects being manipulated as well. For a summary of the information that can be described in various task models, see (Limbourg et al., 2001). The articles (Haan et al., 1991)

and (Kieras et al., 1997) summarise a number of task-hierarchy-based simulated-user analysis models. Some alternative models exist that do not use task hierarchies. For example, Procedural Knowledge Structure Model (PKSM) (Benysh and Koubek, 1993) uses hierarchical flowcharts.

One of the most influential task hierarchy models is Goals, Operators, Methods, and Selection rules (GOMS, see (Shneiderman, 1998) page 55 and (Olson and Olson, 1990) for an overview, (Baskin and John, 1998) for a comparison). GOMS is a well-established method, and many variations exist. GOMS tries to model users' tasks from the 'goal level' to the 'operator level' (which is the lowest level of subtasks). This is done using a set of rules (methods) describing the sequence of steps (operators) that have to be taken to complete a goal, and rules (selection rules) of how to choose between alternative methods according to more specific goal information.

The main idea is that one arrives at a sequence of operators, which are low-level enough to allow prediction of performance (like speed) easily, using easily-obtainable experimental data (for example, duration of pressing a key, typing a word, clicking a mouse, etc.). At this lowest level, a simple cognitive performance model is assumed. Basically, it amounts to summing up all operator durations to arrive at the total duration. Some variations on GOMS have more complex low-level cognitive models, which take into account operators which can be done simultaneously, by using critical path analysis (Gray et al., 1990). Users can be simulated by feeding all rules into an inference engine or an AI engine (for example, SOAR), and then feeding the resulting sequence operators into the low-level cognitive model.

GOMS is typically used for traditional HCI in which the users' goals are clearly defined and the user drives the system. However, it was also used with good effect for a real-time machine-paced interface with machine-driven subtasks (a video game) (John and Vera, 1992), which shows that GOMS is more universally usable. Because GOMS is relatively difficult to use for non-programmers, attempts have been made to make it more usable: NGOMSL (Kieras, 1996) is a simpler, easier to specify variant. Another example is USAGE (Byrne et al., 1994), which automatically generates a NGOMSL specification from a user interface specification created in an UIDE (User Interface Development Environment).

Models vary in their level of detail. A relatively complete and detailed cognitive model, with auditory and visual processor, vocal and manual processor, and computer feedback, is Execute Process-Interactive Control (EPIC) is described in (Kieras et al., 1997). It is based on CPM-GOMS, which models complex temporal dependencies of subtasks, and uses the Model Human Processor model to simulate human behaviour.

There are also automatic methods which are specifically used on NL dialogue. In (Baber and Hone, 1993) and (Hone and Baber, 1995), task flow modelling is used to determine dialogue duration. The emphasis lies on comparing different repair strategies in relatively simple NL dialogues. In their task flow model, statistical word duration, word misrecognition probability and utterance correction probability are used as parameters for a flowchart model of the dialogue, which can then be used to predict task duration. One of their findings was that the best choice of strategy depends on error rate.

Simulation models are especially well suited for the modelling of the efficiency of well-defined routine tasks. Usually missing in these models is a way of accounting for errors and system feedback (Haan et al., 1991). Both may cause the user's subtasks and subgoals to change, especially for tasks where the outcome is not known yet (for example in exploration, when getting to know a computer system, or when searching complex databases).

4.2 Models of the system

The computer part of a setting is entirely under the control of the developer. In theory, everything that needs to be known about the software is already known. Still, models are needed, because of the sheer complexity of most software, making it very difficult to understand. In fact, central in software engineering is the perpetual fight against complexity. In practice, a variety of models is used to describe software. One of these models will be the computer program itself. Other models need not be executable programs, but may aid understanding and evaluation of the software in some ways. The principles of specific types of software modelling are tightly related to the specification languages used, so the finer details are explained along with the overview of specification languages in chapter 5.

Software engineering (SE) is the most important research area that studies the issues of software specification. In SE, the term *complexity* is often used to indicate the difficulty of understanding or building a software system. In fact, one of the main goals of SE is to minimise complexity (Booch, 1994). Software knows little physical constraints, we might say that any software construction can be readily built, with the most important limit being sheer complexity. Just in order to deal with complexity, various special techniques have been developed over the years. When more of these techniques became well-established, more complex software could be built.

One typically distinguishes several software engineering paradigms that have emerged during its history:

- *structured*. Structured programming is the specification of software by decomposing the program to be written into a sequence of subtasks. One can refine subtasks recursively, until one reaches a detail level that is readily programmable. This paradigm is considered mostly obsolete, but it is still useful for writing complex algorithms. A significant disadvantage of this paradigm is that it is not easy to track what happens to the data, as this paradigm does not propose methods for structuring data. What may easily happen then, is that each task in a chain of tasks operates on all of the data, and it is not easy to see what happens to the data after a sequence of subtasks have been performed on it.

- *modular*. A modular program is a program that is explicitly divided into a number of separate pieces, named modules, with explicit connections. Typically, a module specifies what role it plays in the program by specifying its interface, which specifies in which ways the outside world can access the module. Inter-

nally, the module may have an arbitrary structure that does not need to be known to the outside world. This means that one is to some extent able to change the module internally with a minimum of impact on the rest of the software. This manner of separation of 'inside' and 'outside' is called *data hiding* or *encapsulation*. Formally, they prescribe that some part of the software should be declared irrelevant from some designers' viewpoints, and should remain hidden during some design activities.

Modularity is considered an important SE principle. A strictly modular structure in which each module has both explicit and limited interaction with other modules helps in making software development in general more cognitively tractable, for the simple reason that each module can be meaningfully examined in isolation. For instance, testing, extending, rewriting, and maintenance are made easier.

- *object-oriented*. Since the advent of object-oriented programming languages, object-oriented (OO) development established itself as one of the major SE paradigms. OO is a continuation of the modular paradigm, and adds to this the coupling of modules to data types (each module stands for one data type, which is defined by the module's set of functions), the dynamical creation and deletion of items of this data type, and the concept of inheritance. Inheritance is a separate feature, which amounts to creating new data types by inheriting code from other data types, which follows the idea of traditional semantic net classification. Nowadays, it is understood that inheritance is not as important as it was once thought, and it should be used sparingly and with care (Cartwright and Shepperd, 2000).

  Whether the OO modelling technique is actually a good specification technique in general remains an open question. Comparisons of software built using OO with software built using more traditional programming techniques only yield inconclusive results (Smith et al., 1995). Some causes of problems have been pointed out in the literature:

  - OO design invites a fine-grained modular composition, which means that the different modules will be closely coupled. Experiments show that trying to understand the behaviour of a typical object class is a difficult process (Kung et al., 1994). Apparently, this is because it requires understanding of the many other classes that it uses, and the ways in which it is used by other classes. Furthermore, the control flow of the program as a whole is hard to follow, since the locus of control is constantly jumping between objects (Agarwal et al., 1996; Dunsmore et al., 2000).
  - When using inheritance, the behaviour of a class depends on its superclasses. This means that understanding an inherited class requires understanding of its superclasses (Daly et al., 1995). Furthermore, the optimal inheritance structure crucially depends on the nature of the problem domain. If the domain is relatively unknown, inappropriate choices may be made easily (Riel, 1996).

Both the close coupling between modules and inheritance may result in inflexible and hence unmaintainable programs (Kung et al., 1994). Still, the OO paradigm provides a convenient scheme with which to manage dynamically-created data structures, which is one of the things that recurs often in software, and can now be specified comprehensively and concisely.

- *component-based*. Software components are like modules, except that they take the concept of modular decoupling further than more traditional modules. Components are sometimes similar to objects in the object-oriented paradigm, and sometimes have a courser grain, with objects provided as an additional facility. For example, components may be a kind of modules, that exchange objects to communicate.

  Particularly interesting in the component-based paradigm is the decoupling of control flow to such an extent that the components may run as parallel and distributed processes. In terms of distributed systems theory, having each component run as a process is a convenient way to manage concurrency problems, which we will explain in section 4.2.2.

- *agent-based*. Note that we have already defined the term *software agent* in section 2.2.2. The term is also used in SE, but with a more specific meaning. While regular software components are sometimes called agents, usually the term refers to a special kind of component. Generally, an agent is an 'anthropomorphic' component, having the possibility of autonomy in its actions and flexibility in its request handling.

  Meyer (Meyer, 1998) defines software agents as software that is capable of acting intentionally. He proposes that agents should reason by modelling their intentions and those of other agents explicitly. While this is an interesting definition, few concrete systems exist in which agents really do model intentionality to the extent that Meyer does.

  Weigand (Veer et al., 1998, the lecture by H. Weigand) uses a more practical definition: he defines a software agent as a module that is able to use complex communication protocols and has its own database. Such agents make it possible to design systems with a coarser compositional granularity than OO or component systems, since the granularity of these systems has sometimes proved to be too fine. The practice of wrapping a legacy system inside an agent in the hope of increasing maintainability, called 'agentification', is an example of this idea.

  Agents as a software engineering concept are relatively new. It is however not without its problems:

  - It does not seem meaningful to describe a component in terms of complex and possibly vague human metaphors, when the component's behaviour itself is actually clear and simple. For example, the paper (Brazier et al., 1998) addresses an electricity load balancing problem using an agent-based approach. However, one of the most complex issues of this problem is the system's collective convergence to a solution. This issue is actually

addressed using control systems theory. Arguably, the problem was not made much easier by the modelling approach used insofar as it was agent-based: plain control systems theory may have been enough.

– All complexity moves towards the communication protocols and data models, which may be very complex and may still become unmaintainable.

  Some examples of agent communication use Prolog as a communication language. One problem that comes to mind is the complexity of the language interpreter needed. Which Prolog version and engine is being used? What if the Prolog interpreter has to be upgraded, for example, because of fatal bugs?

  Another problem is that processing messages in higher-level languages approaches directly executing code that is supplied by another program. The behaviour of a system consisting of components that communicate by sending arbitrary code to be executed might become very hard to understand.

– The internals of an agent, including any 'flexibility', 'autonomy', or 'intelligence', still have to be programmed.

4.2.1 The User's Virtual Machine

Users, and often the interface designers themselves, typically don't want to concern themselves with the details of the underlying system. Rather, some HCI designers like to talk about the user's conception of the system (which is called the mental model) as fully separate from the system.

For this purpose, a desired mental model may be designed independently from the actual system, according to ergonomic principles. This is called the *User's Virtual Machine (UVM)*. The UVM is not the actual mental model that users have, but corresponds to the mental model that the users should form when using the system.

Various software models exist that formalise this idea of UVM, by defining the place of the functional core of the system and the UVM, and the interaction between them and the user.

A well-known class of models is the Seeheim model and its descendants, which separate the software into a chain of interconnected modules, with at one end the user interaction and at the other the functional core (also called domain model). Some of these decompose each module in further modules. Overviews of these models are described in (Encarnacão, 1997) and (Greg Phillips, 1999). These are sometimes called *Seeheim-like models*, named after the Seeheim model, the first well-known model of this type. Better models have been produced, so we will not discuss the Seeheim model itself. The essential aspects of these models, which are abstract and non-executable, are also called *architectural styles* (Greg Phillips, 1999). Some of these models do have explicit implementation support in the form of programming languages or software libraries, filling in the details not specified by the architectural styles themselves.

Most models have a separation into four or five modules:

- *domain component*: the domain functionality, that is, the actual data that the user

operates upon.

- *dialogue controller*: the code that glues the user interface functions to the domain functionality. Note that this module manages the user interface state.

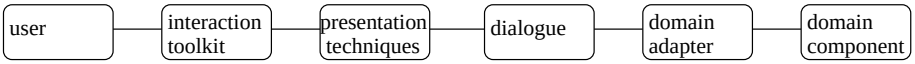
- *interaction toolkit*: the basic user interface functions provided by the operating system.

- *domain adapter*: in some cases, there is a separate module that is situated between the dialogue component and the domain component, which enables the adaptation of the domain model to the user interface to some extent, so that changes in one have less impact on the other.

- *presentation component*: in some cases, another module is situated between the interaction toolkit and the dialogue controller, serving an adaptation purpose, similar to the domain adapter.

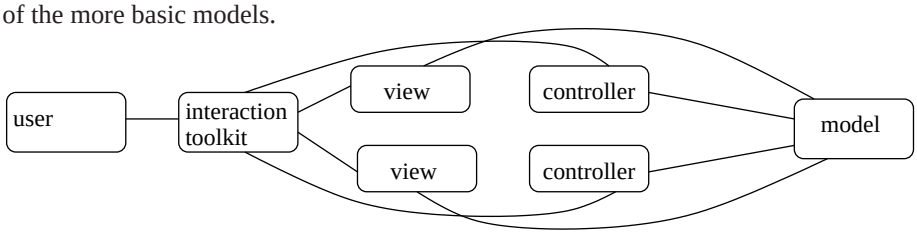
They provide general guidelines for the structure of software systems, which we find reflected in various modelling techniques. Specification techniques that are meant for producing such models are discussed in section 6.3.3.

We will give a short summary of some of the most important of these models here.



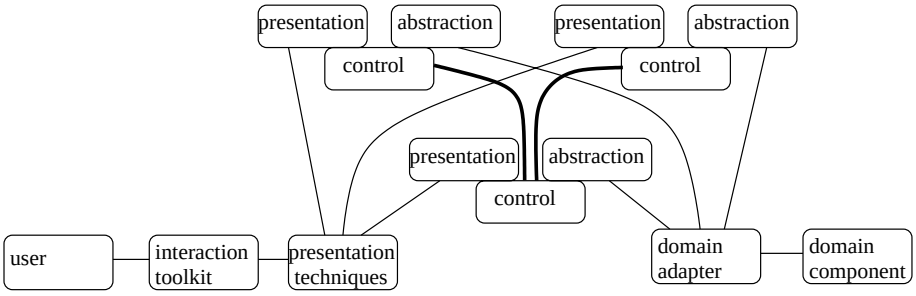
The Arch model

The well-known Arch model is an improved version of the Seeheim model. This is one of the more basic models.



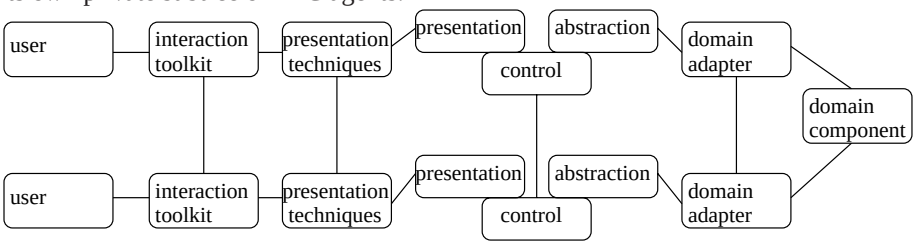
Instance of MVC with one model and two view−controller pairs

The MVC model (Hussey and Carrington, 1996a) is a popular model that may be seen as a concretisation of the Arch model by further decomposing a system into modules. There may be any number of Model components that stand for the domain functionality. Each model may have one or more View-Controller pairs attached to it. The View is notified when the model changes, and updates itself accordingly. The Controller is the module that receives information about user actions and updates the Model accordingly.



Instance of PAC−Amodeus with three PAC−agents

The PAC-Amodeus model (Nigay and Coutaz, 1997) is an explicit extension of the Arch model, in that it specifies the decomposition of the dialogue component into *Presentation-Abstraction-Control (PAC) agents*. The Control module of a PAC agent glues the Presentation and Abstraction modules, and communicates with other Control modules. The Presentation module handles the connection with the presentation techniques module, and the Abstraction module with the domain adapter. The control modules are always arranged in a tree, with each Control module communicating with its own private subtree of PAC agents.



Instance of PAC* with two users and only one PAC* agent

The PAC* model (Calvary et al., 1997; Coutaz, 1997) is an example of a model that is meant for multi-user interfaces. There is a separate PAC-Amodeus model for each user, except that the domain component is shared. Furthermore, the peer components at each level in the chain may have communication. For the PAC agents, this means that each Control module may have communication with any other Control modules of the other users.

In practice, the Controls of different users need to communicate in rather arbitrary ways. In spite of the strict tree-structured communication of the PAC-Amodeus model, no appropriate architectural guideline is maintained for the inter-user communication. This is a serious issue that may be generalised to other kinds of UIs.

This brings us to some problems we may identify with architectural styles.

- In some UIs, UI state information is inherently non-localised, and is difficult to encapsulate meaningfully into a strict architectural style, such as the PAC-Amodeus tree. This particularly goes for multimodal and multi-user systems, in which one party needs to keep track of the various activities of another in an integrated manner. Examples are: a multi-user scrollbar, which keeps track of the scrollbar positions of the other users, ‘radar’ view showing an overview of users’

activities, as for example found in GroupKit, and context-sensitive assistance, in which an assistant agent keeps track of a user's various activities in various parts of the UI and offers help when appropriate. In fact, some multi-user toolkits, such as the popular GroupKit (Roseman and Greenberg, 1997), maintain a separate distributed shared dataspace to model parts of the UI state that should be shared between users. In the case of GroupKit, the structure of this dataspace actually bears no relation to the structure of the UI. In fact, it can be safely said that the dataspace simply bypasses any architectural style used by the UI toolkit.

- While the separation of UI and functionality in the software is a often-mentioned principle, and is reflected in various modelling techniques, in practice there are sometimes inconvenient dependencies between the modules. For example, Evers (Evers, 1999) discusses some examples where this principle is necessarily violated. A particularly important example is that information about the range of the values in the domain must be known to the UI. For example, when a certain field to be filled in has a limited, enumerable range, a good UI choice would be to use a list of all possible options. But then, the range of possible values, which is knowledge normally attributed to the domain component only, should also be known in the UI. This creates unexpected dependencies between the two modules. For example, one solution is to have the list coded in both modules separately, which means that a modification to the range would require both lists to be updated.

- Few styles provide proper notations or guidelines for modelling constraints and interrelations between dynamically-created components, such as multiple windows in a single-user application or multiple users.

- Most styles have the difficulty of matching the style's essential features with its implementation model(s). Usually, no abstract or high-level specification is defined for a given style, and instead, one has to extract and encode the constructs which are inherent to the style using specifications in a regular programming language. Some styles do suggest some form of diagrammatic notation, which is however limited to relatively informal sketches (Calvary et al., 1997; Coutaz, 1997). While such notations are apparently useful, the role that they might play in the specification of such systems is usually not discussed. There are some architectural styles that do use a special-purpose programming language to concisely describe the architecture. An example of this is Chiron-2 (Greg Phillips, 1999), in which an architecture is described using an architectural description language.

4.2.2 Communication models and distributed systems theory

In various interactive systems, we often find a large number of modules with complex data flow among them. The user has various means through which to interact with the system, typically producing events from various sources that have to be processed. Additionally, animations and autonomous agent behaviour produces parallel activity

within the system. Additionally, VEs are often distributed systems, because they are used by multiple users. Such VEs are called Distributed VEs (DVEs or DIVEs).

A distributed system is a software system that is distributed across multiple computers. Note that distributed VEs are essentially different from the application of distributed systems that is meant to increase computational efficiency by delegating tasks to multiple processors running in parallel.

Distributed systems are different and more complex than more traditional non-distributed systems. A large and highly technical body of literature exists about distributed systems. Much of the effort is about hiding the most difficult issues behind schemes that make things easier to understand. For example, much effort goes in making a distributed system behave as much as possible as a non-distributed system. To appreciate the issues fully, a very long introduction would be necessary. We will only summarise the main issues and approaches here, see for example (Mullender, 1993), in order to introduce terminology and basic issues to a sufficient extent to understand the models we will encounter here. For example, some specification techniques take specific approaches to this problem, which will be explained in detail in chapter 5. We identify the following issues:

- **parallelism and concurrency.** Distributed systems consist of multiple processes running in parallel. In order to perform system tasks, these processes have to interact in some manner. This means that the execution flow of the system as a whole becomes much more complex as compared to a single-process system. Concurrency bugs, that is, bugs due to unexpected interactions caused by unexpected combinations of execution flows by the different processes are a notorious problem.

- **communication unreliability.** In practice, the communication between processes is unreliable to some extent. Great effort has been made to provide communication links that are as reliable as possible, but in real-life systems (such as the internet), there is always the chance of a link being broken. Systems have to deal with these failures gracefully.

- **communication latency.** There is a delay between two geographically separated computers, which is bound by the speed of light. This means that there is a performance limit to tightly-interlocked processing that is independent of the connection bandwidth or the speed of the individual computers.

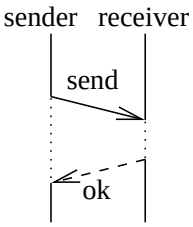
  The main solution to this is not to have processes wait for each other but to have them continue processing independently as much as possible. Various schemes exist for this, which we will discuss in the next paragraph.

There are various schemes to distribute information around an interactive system. We will discuss some of the most common models used here. Firstly, the models may be understood by seeing each module, object, or agent as a separate *process*. This does not mean it actually runs on a different processor, but rather, that it is an entity which has a separate flow of control. We will distinguish three general classes of communication: *message passing*, *shared memory*, and *publish-subscribe*.

Message passing means that processes pass messages to each other. In its simplest form, it is similar to function calling. For example, in the OO paradigm, function calling is by some called 'method calling' and by some 'message passing'. In distributed systems though, message passing is a more general communication scheme than function calling, as it is independent of control flow, and several control flow schemes are commonly combined with message passing. We will discuss four common types of control flow schemes, which we will illustrate with sequence diagrams. In the diagrams, the Y axis denotes time. The vertical lines denote the nodes' lifetime. Solid ones indicate that the process is ready for receiving messages, dotted ones indicate that the process is not ready but waiting or processing. Horizontal arrows denote communications, crooked arrows denote network communication. Solid ones indicate that the communication contains data, dashed ones indicate that the communication is synchronisation without data.

Note that, in the sequence diagrams, the messages take some time to arrive. In some cases, models can be simplified so that message delay is always zero. This is called *atomic communication*, which we will see again in section 6.4.2 on process algebras. In atomic communication, the sender and receiver(s) of a message effectively synchronise with each other. In an atomic communication model, effects of message delays may be modelled by modelling the sending and the receiving of a message by two separate atomic communications with a separate process that represents the communication channel between the processes.

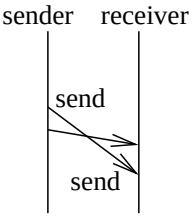
- **Synchronous**.



  In a synchronous model, each message must be handled as it arrives, and the sender has to wait until the message is processed by the receiver. This is the basic model used by most programming languages that know the concept of module.

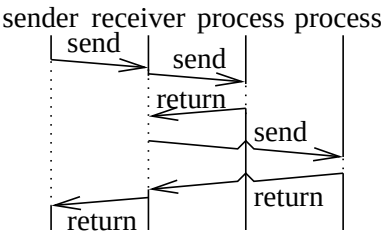
  In the most basic case, each node is reactive, i.e. it handles the message and gets ready for receiving the next message or sending the appropriate output messages immediately. It is possible that a receiver is not always ready for receiving all types of message, and senders may have to wait until it is ready.

- **Asynchronous**.



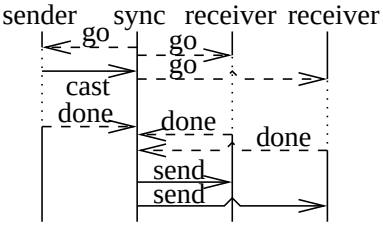
Asynchronous models are useful for distributed communication. In cases where latency is large, it enables a sender to continue processing even before a message it sends arrives. It will typically store its outgoing messages in a local message queue, which is transferred eventually. This means that messages may take arbitrary time to arrive. In the worst case, messages may arrive in a different order than they are sent. Usually though, care is taken that their order is preserved.

- **Serial**.



The serial model can be used when a call results in multiple, recursive calls. This model is the most often used in modular and object-oriented programming languages. We find it in most single-user GUIs, in which messages are generated by the user that propagate through the system. Events are handled one at a time, and each is propagated entirely through all of the processes before the next one arrives.

- **Parallel**.



The parallel model operates in time steps, during which each process may send one message. Conceptually, messages may be generated *simultaneously* in the parallel model, and may be handled simultaneously. In the literature, this model is typically called *synchronous* rather than parallel, but we will use the term *parallel* to avoid confusion with the more common use of the word *synchronous*. While this model is used less often, it is useful for producing parallel activities

that should stay strictly in sync with each other. For example in animation, multiple objects often have to stay in exactly the same relative positions when they are moving together. This may effectively be achieved by the parallel model.

In the figure we specify one time step of an implementation of the parallel model using a 'sync' process that arbitrates system dynamics. First, the sync process signals to all processes that they may generate messages ('go'), after which the processes may do their thing, and signal that they are finished ('done'). Then, the sync node may distribute the messages it received among the processes (in this case one multi-cast ('cast') message), and the cycle ends.

Next to message passing, we have **shared memory**. This means that there is some kind of data structure that the different processes can access, which has control policies built in so that concurrent access always results in well-defined outcomes. Schemes exist that make the shared memory look as much as possible like regular 'local' memory, in spite of network latency between the copies. Examples are replication schemes, in which each process has a local copy that is kept up to date in a certain manner, for example by means of locking schemes or operations transforms. A shared memory system remains, however, essentially different from a regular memory, because changes occur from the outside. Sometimes, these changes have to be known by the rest of the system, and a change should preferably generate some kind of message to make this possible.

An attractive alternative that has such notification built in is the **publish-subscribe** model. It is typically a shared memory in which readers read information by subscribing to it. This means that, every time the requested information changes, the subscriber is notified of this, and can process the changes at will. The publish-subscribe model is usually built on top of underlying message passing or shared memory models. For a discussion of alternative publish-subscribe models, see (Ramduny et al., 1998) or (Patterson et al., 1996).

4.3 Conclusion

In this chapter, we introduced some of the basic models of design knowledge that we will use in subsequent chapters. We discussed models of the user, including executable models that may be used in user simulation techniques, and models of the system, including Seeheim-like architectural models and distributed systems concepts. In particular, we noted some problems with the architectural models that we wish to address in our own technique. These are the following: the problem of encapsulating data while this data is needed in various places, the modelling of dynamically-created components, and the lack of a concise means to implement the style (i.e. a specification language that maps closely to the style's constructs). In chapter 7, we discuss how these problems are addressed in our technique.

Part II

Specification

Chapter 5

Specification languages

5.1 Introduction

In this chapter, we will introduce some theory on specification and specification languages. In chapter 1, we introduced our notion of *specification* as *any* type of description, which is broader than the common notion of *formal specification* which (usually but not always) limits itself to abstract, formal, non-executable requirements specification. We also introduced the notion of *specification technique* as a set of interrelated specification languages. In a specification technique, the interrelation between languages is important as well as the languages themselves. Formal specifications, for example, are not stand-alone specifications, but typically need to be related to executable specifications of the software. As we shall see, this interrelation is not always as well defined as we would wish. How exactly concepts such as 'formal specification', 'executable specification', and their interrelations fit into our model will be made clear in this chapter.

As we will argue, specification is one of the most important activities of software development. With the help of the proper specification techniques, complexity can be reduced. In chapter 6, we will try to shed some light on the alternatives that have been produced in this area. As we will see, SE has introduced a variety of specification techniques for different purposes, some of which can be and are used for VE development. In this respect, SE has made a unique contribution to the area of engineering, as no engineering discipline has produced anything near the richness of manners and styles of specification found in SE. In fact, when we consider the aspect of specification, other disciplines may learn something from SE.

To illustrate the basic issues, we will start with a small example specification that is *not* a typical software specification, but rather, a typical mathematical one. Most readers will probably have seen enough program code and other specifications in the light of SE, and we choose a different area to illustrate that specification principles may also be more generally applied. Consider the following equation, which was taken from my own personal experience, but which we may consider a typically-formulated equation.

$$t(i,j) = c(i,j) \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} f(m,n) cos\frac{\pi(2m-1)i}{2N} cos\frac{\pi(2n-1)j}{2N}$$

This happens to be the Discrete Cosine Transform, such as used by JPeg encoding. This exact formulation was found in several texts on the internet. Arguably, this formula is not easy to read for someone who does not know what it is about. There are various constants and variables in it, *t*, *c*, *N*, *f*, *i*, *j*, *m*, *n*, of which the meaning is not immediately apparent. Typically, one has to spend some time searching for the definitions of these letters in the remainder of the text in order to make sense of the formula. Unfortunately, the authors producing this formula often did not bother to include such definitions.

Yet, it is a more or less established and unquestioned practice of mathematics to use single-letter names to denote variables, and not have a standard way (such as a nice table) to define the meaning of the variables, or perhaps clarify the context in which these variables are defined. In mathematics, one can get away with such an approach because typical specifications are small. Furthermore, our specification above is a variation of the well-known Fourier transform equations. The writer of the specification may have assumed that the reader is familiar with such equations. Another advantage of such a concise specifications is that it is faster to *write*, and we may assume that someone doing mathematical derivations with pen and paper really prefers short variable names.

In software engineering though, such an approach would nowadays be unthinkable. An average program has hundreds of variable definitions. Also, SE often deals with a great variety of subjects, which we cannot assume the reader is familiar with. In the very beginning, software often used single-letter names, or just meaningless number codes, to denote variables. One soon found out that this did not work in the long run, as one kept getting confused about the meanings of the variables. I was quite happy to see that some mathematicians have also adopted this practice. Here is the formula, rewritten using long names:

$$DCTCoef(XFreq, YFreq) = ScaleCompensationConst(XFreq, YFreq) * \sum_{XPos=0}^{NrPix-1} \sum_{YPos=0}^{NrPix-1} PixVal(XPos, YPos) * cos\frac{\pi*(2*XPos-1)*XFreq}{2*NrPix} * cos\frac{\pi*(2*YPos-1)*YFreq}{2*NrPix}$$

This specification is about thrice as long as the first, but arguably it is more readable for someone who does not know its context and is not experienced with such functions. An alternative is a table with variable definitions, which however means looking back and forth between the formula and the table. Another possible advantage is that one makes less accidental errors when working with the specification: since the meaning of each variable is embedded in its form, it is more difficult to make errors.

Using typical software engineering practices, we might generalise repeating patterns in the equation, and give meaningful names to them, to make it even easier to read:

$$DCTCoef(XFreq, YFreq) = ScaleCompensationConst(XFreq, YFreq) * \sum_{XPos=0}^{NrPix-1} \sum_{YPos=0}^{NrPix-1} PixVal(XPos, YPos) * Weight(XPos, XFreq) * Weight(YPos, YFreq)$$

$$Weight(Pos, Freq) = cos\frac{\pi(2*Pos-1)*Freq}{2*NrPixels}$$

This step may make some specifiers unhappy as it somewhat obscures the details of the main equation. The effectiveness of such re-formulation depends on the knowledge and experience of the audience. Yet, such substitution of sub-expressions by single sym-

bols is in fact practiced often as part of proof procedures: one ‘factors out’ a complex part of an equation and describes some properties of it that make it easier to understand. Also, it appears that some mathematicians employ other practices that have been further perfected in software engineering, such as counting the number of symbols in an equation as a measure of complexity.

This small anecdote illustrates some of the basic issues of specification. We already see here some of the basic schemes for reducing redundancy, putting the information one needs in the right places, and providing intuitive context to make interpretation of specifications easier. There are, however, many considerations, which means that often, there is no one ‘right’ way to specify something. Rather, it is often a matter of trade-offs, and this is one of the reasons to use multiple languages in a specification technique.

5.2 The psychology of specification

The software engineering concept of complexity is known in psychology as *cognitive complexity*. While complexity and differences in complexity of different specifications are intuitively obvious in many cases, we wish to stress here that what SE is addressing here is in essence a psychological problem. SE, and, in fact, anyone involved in software specification, is practising psychology, usually without even mentioning the term. Someone working on a software specification will repeatedly ask him/herself questions like: Will I still be able to understand this later? Will others understand it? Will I easily make errors here? Can this be simplified?

Some research tries to explicitly couple software development with psychology, which includes coupling the activity of programming to psychological theory, and trying to measure the psychological aspects of software development by means of psychological experiments. This is usually referred to as the *psychology of programming*. As the name suggests, the area mainly studies programming. While there has been various work done in this area over the years, one may argue that it doesn’t get quite the attention it deserves, if we compare it to the amount of work done in the area of SE that is not explicitly linked with psychology. While much of the work can be generalised to specification languages in general, concrete cases are often limited to executable programming languages. The more general case of *psychology of specification* is largely missing in the literature (Khazaei and Roast, 2001). Relatively much of the work is on visual programming and learning. The psychology of programming has some interesting things to say, which we will try to include in our overview of software specification in the following sections.

5.2.1 Individual differences

There are great individual differences between programmers. A well-cited result is that the differences between programming speed between different programmers may be as great as a factor 20 (Barr and Tessler, 1996; Raley, 1996). The quality of the produced

code of the fastest programmers was certainly not found to be worse than that of the slower ones. Such individual differences suggest that programming is a very complex cognitive task, for which different programmers may have very different personal approaches. Studies of mental models and imagery formed by programmers showed the variety of models that programmers apparently employ (Petre and Blackwell, 1999). A few excerpts:

> "One of the earliest things is to visualize this structure in my head, a dynamic structure, so I can think about how things fit together and how they work ... and once I have the structure fairly strong and clear in my mind, I move it around and move around inside it, examining it and tweaking it ..."

> "[imagery] is a way of harnessing the mental machine for tandem problem solving ... if there's a problem I can't solve ... I get the mental machine to solve a subset of the problem, and I need to see what it's doing in order to understand how that activity relates to the larger problem ... "

> " ... it moves in my head ... like dancing symbols ... I can see the strings [of symbols] assemble and transform, like luminous characters suspended behind my eyelids ... "

These and other examples suggest a very personal imagery, as well as a variation in general problem-solving approach. We also see this in statistical studies, which have attempted to predict the productivity of programmers according to metrics such as 'years of experience' and 'programming languages known'. These are actually significant predictors, but not very good ones (Raley, 1996). Several studies reported that there are great differences between programmers not accounted for by any known metric (Raley, 1996; Mills, 1996). In fact, personal differences make it hard to obtain statistically significant and otherwise meaningful experimental results for more complex programming problems.

Such individual differences are recognised widely (Barr and Tessler, 1996), though some have noted that programming skill is not seriously rewarded in most organisations (Stumm, 1993). In contrast, some organisations have even adopted a 'star developer' based development strategy (Russinovich, 1998), in which the only prescription is that a particularly skilled developer is given the lead in a project. Some people have indicated that there is no real substitute for developers who do not have the required skills, such as Brooks in his book 'the mythical man-month' (Brooks, 1995). In large teams, the need for communication results in serious overhead, which makes a small team of good developers more effective than a large team of average developers.

This suggests that little-skilled programmers are no substitute for much-skilled ones. How exactly such skills emerge is another research topic that is not addressed here. If we create a technique for maximising effectiveness of systems development, we shall argue that we should rather create *tools* for the skilled rather than *remedies* for the unskilled.

5.2.2 Specification language design: a small history

Many schemes have been conceived that augment the syntax of a language in such a way that it is more usable. It should be easy to read and write, and as few errors as possible should be made. The first question here is how we may determine if an expression written down in one language is better than another. *Software metrics* (Kitchenham, 1996), defines some criteria and metrics which enable the evaluation of various aspects of specifications. One very often used metric in determining readability is *conciseness*: the amount of code spent to perform specific functionality. A very often used process-oriented metric is the *time taken* to write a correct version of a specification.

Beside these, the amount of errors made and the time taken to repair them is considered important. While people always make errors, it typically takes a lot of time to detect, locate, and correct an error. Languages may be designed to account for the constant occurrence of errors. For example, the syntax of some languages is designed in such a way, that certain errors cannot be made due to the form of the syntactical constructs, and certain errors made in the specification surface as syntax-level errors, which are then conveniently detected by the language interpreter or compiler. Note that this sometimes conflicts with conciseness. Some error-detecting techniques consist of some kind of double checking, and are at the expense of conciseness. Most of these techniques seem to have emerged through intuition and reasoning, as comparative evaluation of full-blown languages in real situations is too difficult. We might even say that evidence that identifies progress being made in specification languages over the years is not systematically gathered and remains inconclusive in some areas. We will give some small examples of traditional programming language constructs here, and some of the lines of reasoning that led to them, and discuss others in the next chapter, along with the specification languages.

One of the most well-known developments in programming languages is the availability of high-level control flow expressions, enabling the expression of the most common control flow concepts as special shorthands. This technique has a colourful, but not very conclusive, background in the literature. An important thread leading through it is the ‘Goto’ debate, which was sparked by the ‘Goto considered harmful’ paper by Edsger Dijkstra (Dijkstra, 1968), which did not really provide conclusive evidence that Goto is, indeed, harmful. In fact, neither did reply papers such as ”‘Goto considered harmful’ considered harmful”, etc. (Rubin, 1987; More, 1987). The evidence given is mostly anecdotical, intuitive, or illustrated by example programs. The examples often serve to compare one program segment to an alternative one, and it is assumed that the reader will know which one is the most preferred one. In some cases, the conciseness metric is used.

Eventually, such reasoning has led to Goto being gradually replaced by a number of high-level constructs, such as while, for, and throw-catch. In several common languages, such as Modula 2 (Sutcliffe, 1987) and the recent Java (Zukowski, 2002), Goto was left out entirely, with the authors claiming it was considered harmful. In retrospect, it remains somewhat questionable whether use of Goto makes a real difference compared to other issues. For example, modern developments in software have introduced new control flow issues, which may be considered even more ‘harmful’. One such ex-

ample is concurrency. Concurrency is notoriously difficult, yet the wide availability of it in modern systems has led to people using it gratuitously, which may be the cause of many bugs. Another control flow issue is introduced with the throw-catch exception handling construct: while typically advocated as 'high-level', it is also difficult to use correctly (Reeves, 1996).

This does not mean that the concepts of concurrency and exception handling are not useful, but rather, that it is easy to misuse if you don't know what you're doing. Continuing in the same vein as the Goto, we may argue to forbid concurrency and exception handing as well. Actual *forbiddance* of such constructs is an example of a remedy for the unskilled: we forbid it in case it *might* be misused. However, we retain here that a tool being difficult by no means implies that it is harmful. In some cases, describing control flow in terms of Goto may be quite useful. In fact, one of the solutions to control flow problems is creating an abstract model of the particularly difficult program segments by means of state automata–which is similar in structure to control flow described by means of Gotos. Some have even suggested that the Goto has become a taboo, which is related to the impression that 'Dijkstra is considered holy' (Marshall and Webber, 2000).

Another example of error-reduction by means of language constructs is type checking. As regards type checking, it is widely believed that rigorousness of typing and type checking is positively related to the reliability of the resulting programs (Meyer, 1997), as type errors may point to conceptual errors. Yet, various popular languages exist that apply everything from very little to very much type checking, and each of them seems to have its own advantages and disadvantages.

Another is the format of 'if' statements. (Sime et al., 1977) describes an experiment in which two different specifications are compared with respect to the speed with which errors can be found. The specifications were relatively simple: both specified a number of conditional ('IF') statements. The only difference between them was that one of them specified the negative condition explicitly. As it turned out, the version with the negative conditions improved the speed with which errors could be found.

5.2.3 Discussion

We indicated that current specification language design practices may benefit from a more careful examination, but that it remains a difficult endeavour for which no systematic process or theory exists. Obtaining experimental evidence is difficult, and is limited to simple cases, though work is still being done to improve experimental methods (Douce, 2001; O'Brien et al., 2001). While the tools provided by the psychology of programming and by software metrics will be some help in our first step towards a VE specification technique, we should be cautious as to drawing conclusions.

In our case, the best tool we can use here is careful reasoning combined with ample examples, in which case we are required to trust our understanding of the subject matter. Our understanding can be improved by existing psychological theory from the psychology of programming, explained in section 5.3. Furthermore, we shall describe criteria that cover the usefulness of a specification language in section 5.4.

5.3 Psychological theory

Some psychology of programming theory has been developed, mostly based on cognitive psychology. We give a short overview of existing theory in this section.

5.3.1 Specification and memory

One of the most important aspects of the psychology of programming is the theory concerning human memory. There is psychological evidence that experienced programmers have a good memory for programs, being able to memorise relatively complex programs within short-term memory (Soloway and Ehrlich, 1989), just as experienced chess players are able to quickly memorise an entire chess board. We may assume that, like chess players, programmers must have often-occurring programming patterns as memory chunks. The studies of Soloway et al. (Soloway and Ehrlich, 1989) indeed show that atypical programs are more difficult to memorise by experts than typical programs. They hypothesise the existence of *plans* (often-occurring solution patterns, somewhat like design patterns), and *rules of discourse* (conventions about how a program should be made easy to read, such as variable naming conventions) in long-term memory.

Since human memory is too limited to memorise a software system in its entirety, understanding software is closely coupled with reading specifications. Programmers typically spend a lot of time looking at code. Davies (Davies, 1993) calls this *display-based reasoning*: the display of code or perhaps other specifications is used as a memory aid while reasoning about the software. It appears that experienced programmers even practice *more* display-based reasoning than beginners. One theory is that people comprehend specifications by finding *beacons* (Crosby et al., 2002): constructs which look familiar, and hint towards some often-occurring solution. The concept of *beacon* appears to be related to the concept of *plan*. For example, a programmer sees a program segment consisting of three assignments, one of which is an assignment to a variable named ‘temp’ (a beacon), and infers that a variable swap is occurring, and concludes that s/he must be looking at some kind of sorting routine (a plan).

Taking this a level further, it is likely that programmers, when imagining the part of the software that they do not yet understand, are continually assuming the simplest and most common solutions, possibly at a conscious, rational level. This principle is also called *Occam’s razor*, or *parsimony*. Combined with the concepts of plans and beacons, parsimony may be used as a basic rule of communication when reading and writing specifications. For example, consider a simple loop that processes n items, which counts down from the nth to the 1st item. When someone sees this loop, it is likely that s/he will wonder if there was a special reason why this loop was specified to count backwards rather than forwards, which is the most obvious manner to specify the loop. There are various conditions in which misunderstandings may occur, for example, counting backwards is sometimes done for efficiency reasons, and someone who does not know about this may be puzzled about backwards-counting loops. A programmer who understands the possibility of such a misunderstanding is likely to

add a comment, stating the reason for counting backwards.

5.3.2 Style and language

As stated by the theory of distributed cognition, the specification language shapes the programmer's thinking to a great extent. For example, several people have reported that programmers reproduce program code literally in their imagination (Weinberg, 1971; Petre and Blackwell, 1999). Also, the well-known quote *'some people can write <LANGUAGE> in any language'* suggests that each language has a different style associated with it, and that use of a language does not imply that that style is indeed used. Styles adopted from experience with one language may be used when specifying in another language. If a language was not made for a specific style of specification, it typically means that such specifications are still possible, but they become more difficult to read or write (Green, 1990a).

Specification languages and styles are implicitly linked with theories and guidelines about design. For example, the Goto debate is part of the structured programming movement, and the use of high-level constructs in structured programming may have influenced people's thinking about control flow. So, when people who have grown up with structured programming concepts are confronted with a language that only provides Goto, they will inevitably use it to encode the high-level constructs they are familiar with. It is uncertain what happens if these constructs are *not* known. In some cases, programmers may invent their own 'high-level' constructs as soon as they find out that a 'low-level' construct is difficult to use. The real strengths and weaknesses of a specific technique will be known only after a sufficient amount of specifications were made.

Some *a posteriori* studies exist that try to show that some program constructs are almost always used in a limited number of ways. An interesting example is the *variable role* concept. Studies indicated that almost all program variables can be classified as instances of a limited number of roles (Sajaniemi, 2002). The following roles are distinguished: fixed value (i.e. a constant), stepper (i.e. a loop counter), most-recent holder (holding a value that has to be processed in the current loop step), most-wanted holder (holding a value indicating some desirable property, such as the highest or lowest number found in a loop), gatherer (gathering cumulative results during a loop), transformation (a temporary value that is calculated from other values according to a fixed calculation), and follower (a variable that follows the value of another variable from a previous step). It seems meaningful to provide language constructs for such high-level concepts, making their distinction clear at a syntactic level. In fact, some of these roles are already covered by syntactic constructs, in particular the fixed value by a *constant* construct, and the stepper in loop constructs. In our assessments we will also pay some attention to such *a posteriori* specification concepts.

5.3.3 Tasks and specification languages

It appears that some languages (in particular abstract, diagrammatic languages) are formalisations of what programmers produce spontaneously when not given any specification guidelines. Apparently, there are good reasons why these languages exist: possibly they make certain software development tasks easier. There is in particular strong evidence that dataflow diagrams are one such kind of language (Petre and Blackwell, 1999). This indicates that programmers produce mental models in terms of other languages as well as the programming language. This is not surprising, as it appears that different languages emphasise different aspects at the cost of others (Green, 1990a).

Generally speaking, we may see a specification as an information structure which is traversed by the programmer according to the tasks at hand. Theoretically, one could follow programmers' typical tasks to see what kind of specification would be an optimal information structure for each task. One distinction made in the literature is between 'programming' (a kind of design) and 'debugging' (a kind of analysis) tasks (Green, 1990b). The one more often requires an overview, while the other more often requires the finest detail to be inspected at will. Another distinction made in the literature is the classification of software into common aspects which are typically needed to comprehend it (Green, 1997). The aspects most commonly identified are control flow and data flow.

A basic principle of specifications is that they should enable the designer to see and hide what is desired at any moment. Different tasks require different emphasis. This can range from something as simple as being able to temporarily replace a part of the design by a single box or piece of text, to being able to view a design from different complementary viewpoints.

Note that this is sometimes in conflict with the software engineering principles of data hiding and encapsulation. More specifically, the 'hidden data' should not be necessary because there are other specifications that should provide the necessary information. This principle is not always maintainable in practice, as in many practical situations, this 'data' actually contains relevant information not found in any other specifications. This occurs for example in debugging, but it also occurs if the interface is under-specified and additional information is required about a module's functionality. This happens regularly, and this is one of the things that the well-known programmer's sayings *'the source is the documentation'* and *'use the source, Luke'* refer to.

5.3.4 Writing specifications

We may specifically look at program creation rather than comprehension: how do programmers make decisions when writing programs?

One theory that addresses the difficulty of the solutions that programmers write is *attention investment*: the trade-offs that any programmer continually makes in choosing a lazy way or a more complicated, less lazy, way of doing things (Blackwell and Green, 1999; Blackwell, 2001). An example of an attention trade-off is the choice between trying to find a function that does what you want and writing some code to do it your-

self. Another is writing a new function versus repeating similar code multiple times. The theory suggests modelling choices by labelling each option with potential value, risk of failure, and cost. However, attention trade-off is not necessarily a rational activity, with the developer making quantitative attention trade-offs explicitly. For example, there is an *emotional* resistance against dealing with complexity, a kind of 'mental laziness'. Cooper (Cooper, 1999), for example, introduces the concept of *cognitive friction* which is such a kind of emotional resistance, and also points out that the level of resistance tolerated differs for each individual. In particular, people who suffer less from such emotional resistance are called 'inmates'. Cognitive friction has significant impact, and may be considered an important additional factor in attention investment. For example, people often prefer a lengthy but easy to do menial job over a job that is more reliable and/or takes less time, but which requires concentration. An example is doing a search-and-replace by hand rather than using the appropriate built-in function.

There is evidence that programmers use solutions that are well known to them rather than make use of available tools (Schank and Linn, 1993). This is quite understandable if we consider that, for example, software library functions turn out to be difficult to understand and use correctly (Scholtz, 1993). Perhaps these functions are not as simple as people who are used to them seem to think: there is considerable attention investment involved. Likewise, people appear to prefer specification styles that are the best known to them when first using a new language, even if this produces cumbersome specifications. People might for example use only a subset of the features of the language. It is likely that programmers need a lot of time to consolidate the details of new features into their long-term memory before they can use them effectively. There may be another reason for the systematic misestimation of this difficulty: which is, that humans tend to underspecify without being aware of it (Levinson, 1987).

5.4 Suitability criteria

Each language in a specification technique may correspond in some way with other languages in the technique, and have various degrees of suitability in various aspects and contexts. We define four general criteria here: well-definedness, expressiveness, cognitive tractability, and computational tractability.

Note that these properties are similar to the criteria found in the taxonomy of Brun et al. (Brun and Beaudouin-Lafon, 1995). In this paper, a similar overview and evaluation is given, though more emphasis is placed on HCI specification techniques, and less on software engineering and AI techniques. The article distinguishes 12 criteria. The first 10 are categorised in two classes: expressive power (similar to expressiveness) and generative capabilities (similar to computational tractability). The other two are extensibility (which does not have a separate entry here, as we consider this out of our research focus), and usability (similar to cognitive tractability). We will return to this taxonomy in more detail in the next sections.

A second related approach is that of Green's *cognitive dimensions* framework (Green, 1989), which in fact covers cognitive tractability. Here, a number of general cognitive

properties that can be attributed to specification styles or languages are identified. We will also give a short account of this framework in the next sections.

A third related approach is the set of criteria introduced by Meyer (Meyer, 1985), which are defined in the context of formal versus informal requirements specifications, and are meant to assess the formal preciseness of a specification. These criteria, called *the seven sins of the specifier*, concern well-definedness. They are noise (irrelevant features of a specification), silence (omission), overspecification (suggesting an implementation rather than stating the constraints of the problem), contradiction, ambiguity, forward referencing, and wishful thinking (features that cannot readily be implemented). These criteria are easily enough understood as basic well-definedness criteria, with the notable exception of overspecification. It is not really clear to what extent any specification suggests a specific implementation. Say, if we specify a mathematical formula, such as our JPeg formula, do we not suggest a certain order in which the terms in the formula should be multiplied? Meyer shows that using a formally defined language (i.e. a well-defined language) mitigates these problems.

5.4.1 Expressiveness

Many languages have limitations to their expressiveness, which means that it is theoretically impossible to fully describe some systems that can be described using other languages. Expressiveness weaknesses in one language may be complemented by another language. The fact that a system may be expressed by means of a particular language does not necessarily mean that the model thus formed is actually useful. In some cases, we may theoretically specify some systems, but the models thus formed become so cumbersome that they are practically useless. In this case, we consider the language to have limited practical expressiveness.

Languages with limited expressiveness have reason to exist because they have a higher tractability than other, more expressive, languages. Examples are various diagrammatic languages, which are little expressive, but form a cognitively tractable complement to the high detail level of programming languages, which are among the most expressive.

5.4.2 Well-definedness

Intuitively, each specification language that is part of the technique should best be highly unambiguous. Furthermore, its relation to the setting under development and the other languages should be well-defined. A language that is or can be unambiguously defined by means of a formal semantics, coupling it to something well-understood, such as predicate logic, may be called a *formal specification language*. In this light, a programming language is a kind of formal specification language too, since expressions in a programming language (should) unambiguously define computational behaviour.

A specification's relation to a setting may be described by the concept of *completeness* or, conversely, *abstractness*. The more complete a specification, the more well-defined. While completeness is an intuitive concept, it is not always easy to find out just how complete a specification is with respect to something else. When one considers the

full setting including users and computers, any specification of it will necessarily be incomplete as much as it will be ill-defined in other ways. When one just considers the software though, one can more easily define how complete a specification is. Intuitively, one might say that program code of the full working software is a complete specification of the software. Still, the story is not quite that simple, and this manner of completeness is not quite a formal concept. Intuitively, one still tends to separate intention from implementation, even if intention is not specified explicitly. For example, many program behaviours are immediately and unquestionably identified as 'bugs'. In fact, we may go a step further, and say that any software that contains errors is not well-defined. Complementary specifications that are used to formally verify the correct behaviour of a program are not complete either, neither do they have to be correct. If viewed this way, we can say that no complete specification exists of the software system either.

A language may be meant for more or less abstract specifications. The reason for the existence of 'abstract' languages is typically higher tractability. For example, programming languages often express a level of detail that may be considered too high to be tractable for some kinds of formal analysis, and it is still useful to have additional, more abstract, specification languages. A high-level programming language, such as functional languages and Prolog, may be seen as lying on the border between a typical abstract formal specification language and a programming language, as they are more tractable to formal analysis, and at the same time, they may be less suitable for being part of the final executable system. Sometimes, abstraction is explicit and deliberate (such as a diagram representation of an existing program) but sometimes, it is done implicitly or even unwittingly (such as a typical natural language specification of a desired system).

5.4.3 Computational tractability

One cannot write down a large formal specification without making serious mistakes that are very hard to detect without actual testing or other computational feedback. Any programmer knows that it is not really possible to write a large computer program without testing versions of it regularly while the program is under construction (Green, 1990a). Worse, people even have trouble writing small specifications correctly (Gurr, 1994; Stenning and Gurr, 1996). This means that a specification language that is executable has a great advantage over one that is not. Executability is an important kind of feedback, but there are others. We identify five types of computational feedback here:

- *Syntax and type checking.* This is the ability to check if any specification is correct at the surface level only. Examples are syntax checking of logical equations or of diagrams. This may be considered one of the lowest levels of computational feedback.

- *Assertion checking.* This is verification of properties by checking these properties for each specific case encountered. This is now reasonably common practice in programming. When the program runs, any assertions defined are checked

for each state the program is in. This is by no means an exhaustive check of the correctness of the program, yet it is a great help in locating errors when they surface.

- *Theorem proving.* This is the ability to feed equations into a software tool to enable automatic or semi-automatic proofs. One distinguishes semi-automatic theorem proving (which is sometimes called *proof assistance*) and fully-automatic theorem proving.

- *Simulation.* This is just the ability to execute a specification as a software program. This includes simulation of abstract specifications, such as state automata, or execution of full-blown software.

- *Model checking.* This is the ability to exhaustively check a specification for correctness criteria. While progress is being done in this area, usually there are severe limitations to the size of the specifications being exhaustively checked. It is only applicable to abstract models.

5.4.4 Cognitive tractability

Cognitive tractability means that a specification should be readable and writable for the relevant parties (which may include users as well as designers in some cases), and as many relevant aspects of the problem as possible should be easily expressible, and obvious upon reading. As we have stated, different languages are typically used, each meant to emphasise a different aspect that is not well emphasised by another.

Note that, according to this criterium, not all descriptions are actually suitable as specifications, even though they may be specified in a specification language. In many cases, descriptions in one specification language are actually generated from specifications in other languages. These descriptions are often highly unreadable, and there should be no need to read them. We consider these specifications to have zero cognitive tractability. In some cases though, the generated descriptions *are* useful as specifications, as they provide a different view of the specifications in the source language, and we may call such specifications *view specifications*. The writing process is however different: any changes that need to be made, even if prompted by the view specification, must be made in the source language. This means that there should be a comprehensive mapping between view specifications and source specifications.

We will review some properties related to cognitive tractability here:

- declarative...procedural

  This distinction between declarative and procedural specification recurs often in both cognitive psychology and software specification. The distinction is a distinction between ‘what’ versus ‘how’, or:

  1. the distance between the specification and a step-by-step description of what to do, or

2. more generally, the explicitness of implementation details.

Procedural specifications are generally more easily executable, as executable declarative specifications create trouble in their unintuitive mapping from specification to step-by-step execution. Generally speaking, the efficiency of executing a declarative specification often remains a surprise.

- Green's 'cognitive dimensions' (Green, 1989) which are in effect evaluation criteria usable for any kind of authoring system. This is one of the best-known set of criteria. The criteria are not formally defined nor part of a rigid methodology, but basically, they extend the vocabulary of the developer and user of specification techniques. We will briefly list the criteria here.

Viscosity	resistance to change
Hidden Dependencies	important links between entities are not visible
Visibility and Juxtaposibility	ability to view components easily
Imposed Lookahead (premature commitment)	Constraints on order of doing things
Secondary Notation	extra information in means other than program syntax
Closeness of Mapping	representation maps to domain
Progressive Evaluation	ability to check while incomplete
Provisionality	degree of commitment to actions or marks
Hard Mental Operations	operations that tax working memory
Diffuseness/Terseness	succinctness of language
Abstraction Gradient	amount of abstraction required, amount possible
Role-expressiveness	purpose of a component is readily inferred
Error-proneness	syntax provokes slips
Perceptual mapping	important meanings conveyed by position, size, colour etc
Consistency	Similar semantics expressed in similar syntax

The meaning of most terms should be reasonably clear from the table, though good use of them would of course require examples and practice. Abstraction is not the same as abstraction as we have defined in section 5.4.2. In the cognitive dimensions framework it means: the ability to define new concepts that can then be used by just referring to them. Note that the aspect of secondary notation is covered here by the notion of multiple specification languages. Consistency is similar to consistency in usability.

5.4.5 Content and process suitability criteria

Parallel to the four criteria defined in the previous sections, we may identify another two criteria, based on our summary of theories and techniques in chapter 2, 3, and 4. These two criteria are more concrete than the four relatively generic ones, and they couple languages to a more concrete and specific context.

- **Content suitability** is suitability for specifying certain aspects of certain kinds of systems. This refers to the properties defined in our characterisation of VEs (chapter 2), as well as the general specification aspects found in methodologies, defined in section 3.7.

  There is a great variety of languages that may cover such content, with various levels of expressiveness. Note that the same language may be employed for different kinds of content. The matter of coupling the language to the real world is a matter of well-definedness.

  Note that the Brun et al. taxonomy we mentioned at the beginning of the section also covers a number of expressiveness properties which are related to specific content:

  The user's tasks and actions.
  The interface state and system's feedback
  The sequencing of actions including time constraints
  The parallellism of actions
  The presentation of the interface
  The management of user errors

  While these aspects are primarily interface- and task-oriented, they do have some overlap with the content we have identified in section 3.7. Suitability with respect to the specific VE properties is more complex, and will be discussed in chapter 6, along with the way the different languages are actually used.

- **Process suitability** is suitability for certain kinds of development activities. The relevant kinds of development activities are discussed in chapter 3. If we look again at the Brun taxonomy, we find that it lists a few process suitability criteria as well. They are in the computational tractability category, enabling computational feedback in the development process. It distinguishes the following, called 'generative capabilities':

Conduct a predictive analysis	
(e.g. cognitive load, task time execution)	simulation
Prove a system's properties	theorem proving
(e.g. termination, accessibility)	or model checking
Generate all or part of the final system	
(e.g. code generation)	simulation
Derive generic interface functions	
(e.g. contextual help, cut and paste)	simulation

  The first two are very clear examples of process suitability. The last two are only related to creation of the final system. This is a nice start, but looking back at chapter 3, and especially at figure 3.1 we may identify more general process suitability capabilities:

  – Readability for non-developers, which is useful for user-centred design and specification walkthrough
  – Correspondence of models with qualitative analysis models, for example, they could be written in the same languages. This is useful for any devel-

opment including qualitative analysis, and possibly, also for systems that incorporate user modelling.

- Generation of prototypes or WOz simulations.
- Usage in user simulation models, which may enable conduction of predictive analyses.
- Sanity checks, including proving a system's properties, assertion checking, and syntax checking

5.5 General types of specification

In this section, we identify four major approaches to specification, according to the main purpose of the specifications, and locate them in terms of our criteria. A specification language may be categorised as an instance of one of these four approaches.

- **Simplified system simulation.** Such specifications are underspecified with respect to the final system, yet result in an executable system. They are mostly procedural.

  Some of these notations are simply programming languages, tailored for interaction design. The languages may be very limited to keep them simple, but they are still useful to create an approximation of an envisioned system which allows early usability analysis. The programs written in these languages are either limited to simple prototypes, or are not quite fit for producing professional products in other ways.

  Another class of such notations are notations meant to evaluate correct behaviour of a simplified model of the system, usually without emphasis on the human-computer interaction. Some of these specifications are model-checkable, so they enable extensive correctness checking.

- **(high-level) programming.** Such specifications directly describe the final system, which may be tailored specifically for the class of systems being built. They are mostly procedural.

- **Abstract models, or intuitive models.** Such specifications are not necessarily formal, nor executable, and are generally much underspecified. They typically have a high cognitive tractability, and some of them require less computer expertise than most other specifications. Good examples are sketches and diagrams.

- **(Correctness) constraints, or 'formal specification'.** Such specifications specify system properties in a formal, declarative way. The specifications are typically underspecified, which is sometimes desirable. This enables a range of possibilities to be specified, explicitly leaving some decisions open, in the sense of Meyer's concept of overspecification. They are usually not executable, but they should preferably be tractable enough for assertion checking or theorem proving.

The four approaches are further illustrated in figure 5.1.

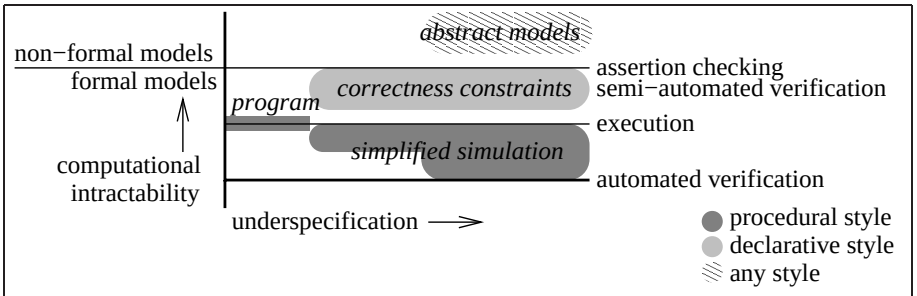


Figure 5.1: A sketch of the computational dimensions 'computational tractability' and 'underspecification'

5.6 Conclusion

In this chapter, we presented an overview of the theory and practice of specification language design. We decided on the following things:

- We emphasise tools for the skilled rather than remedies for the unskilled.
- We distinguish specification style from specification language, and must try to account for the correspondence between these two.
- We require a sufficient amount of example specifications to illustrate the suitability of a specification language.

Furthermore, we introduced four general language suitability criteria well-definedness, expressiveness, computational tractability, and cognitive tractability. Next to that, we discussed development process specific and content specific criteria.

Chapter 6

An assessment of specification languages

The goal of this chapter is to provide an overview and assessment of specification languages that are useful for specifying the different aspects of VEs. The assessment can be used to create a new specification technique that covers the different requirements inherent to the full set of VE properties we have identified. In our approach here, we emphasise the aspects of well-definedness (its relation to the final system) and computational tractability (the computational feedback tools available) to an extent that is more than typical, which leads to a different perspective on some languages.

We will present a classification of specification languages. While this classification covers a rather broad range of languages, it is not meant to provide some kind of exhaustive coverage of language types. Rather, we only cover languages that are well established, seem particularly useful, and/or have been used specifically for VEs. In fact, almost all of these languages are used for development of VEs or applications related to VEs, and we do cover most of the high-level VE toolkits found in the literature.

For each class of languages, we will give a short account of its application area and main purposes, and give one or more examples. We will discuss some existing variations, and the differences between them. We will give a short account of the languages' suitability for VE specification using the suitability criteria defined in chapter 5. While expressiveness, well-definedness, and computational tractability are relatively easy to determine, cognitive tractability is more difficult, and we will refer to literature where possible, and use the given example(s). We will conclude with an overview of the languages and their suitability, that will naturally lead us to the introduction of our own specification technique in the following chapters.

We will classify existing languages according to the specification styles that characterise them. We distinguish three styles: logic, structure, and control flow (see figure 6.1). Each style has a certain purpose, certain formal characteristics, and a certain intuitive background. Each language is to some degree inspired by one or more of these

styles. With some languages this correspondence is very clear, with others it is more difficult to identify. We will try to indicate how the different languages are related to these styles. Note that at some perhaps surprising points, the styles and languages overlap. We will try to indicate when languages that seem very different have great similarities.

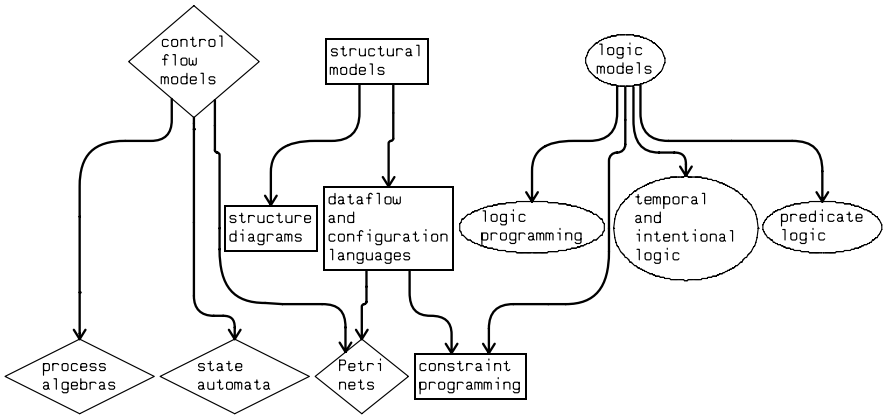


Figure 6.1: Conceptual correspondence between languages
The circles indicate languages discussed in section 6.2. The boxes indicate languages discussed in section 6.3. The rhomb shapes indicate languages discussed in section 6.4.

We use the term *logic* to describe various logical, arithmetical, and computational expressions in a traditional mathematical manner. Logic can be seen as one of the bases of programming languages, since programming languages describe logical expressions in detail. There are also special logic languages that enable some form of logical deductions to be made from a set of logical rules.

Some languages focus on *program structure*. This structure may be considered independent of other aspects of a language. The very same behavioural specification written in the same style may be structured in many different ways. The difference is often just cognitive tractability. Beside that, the introduction of structure is often meant to reduce redundancy, that is, to prevent having to specify the same thing twice. Various aspects of software may be structured. For example, there is data structure and program structure. In some languages, these two structures are meant to co-incide.

Control flow describes aspects of computation, as logic does, but from a different perspective. It is mainly concerned with the description of dynamics, typically by identifying a number of states that the system can be in.

A similar classification is the Brun et al. taxonomy (Brun and Beaudouin-Lafon, 1995). Here, a number of specification languages are classified by their origin, rather than their nature: cognitive science, calculus theory, and the theory of categories. The emphasis there is on abstract languages, including task description languages, control flow lan-

guages, and logic models. The languages deemed worth mentioning are a number of task modelling languages, Petri nets, object-oriented languages, state automata, process algebras, and logic methods. Except for the task modelling languages which we do not focus on here, we will cover all of these languages here, and more. The only exception to this are object-oriented languages, which have been discussed in chapter 4 along with the common programming paradigms, and need not be covered here.

6.1 The running example

We have already mentioned the complexities involved in evaluating a specification technique, and the necessity of examples. We will make a point of giving examples here. While it is not always easy to specify a comparable example in very different languages, we will use a running example here to enable some level of comparison. Sometimes, we will only specify aspects of this example. Next to the running example, we will use various smaller examples where appropriate, some taken from the literature.

The running example is inspired by the Virtual Music Centre (VMC) system, which we introduced in section 2.3. It contains each of the aspects of a VE as we have defined: multimodality, a dialogue agent, multiple users, and graphical interaction. We will call this system *Mini-VMC*.

Mini-VMC is a VE, situated in a virtual version of the building called the **music centre**. *It is combined with a Web environment consisting of a number of HTML pages with related information. Multiple users may log into the VE. Users present themselves using avatars, and can talk to each other in chat room fashion. There is a dialogue agent situated in the VE, which can be asked about musical and theatre performances, and allows reservations to be made. It tracks the users' pointing at objects in the VE, and their Web browsing behaviour, and answers queries about theatre performances. It may also show a list of query results, which are displayed in a table. Results may be selected by clicking on them.*

6.2 Logic models

We define *logic models* as models that describe logical, arithmetical, and computational expressions. There is a continuum between these different kinds of expressions: for example, an assignment statement is a computational expression, which may be in some cases directly translated from an arithmetical expression. Logic models may be more or less procedural, and either directly describe a sequence of operations, or describe a number of constraints or inference rules which stand for a space of possibilities and can be used as correctness constraints. The procedurality of a logic model is intimately related to computational tractability. Procedural logic is found in almost any programming language, more declarative logic is found in some special languages, some of which we will discuss in this section.

We assume that the reader has some basic knowledge of logic. We will not go into detail about the formal characteristics of different logic models, but this should not preclude making some general observations about languages based on logic models. In short, we may distinguish several kinds of logic: the most basic is *propositional logic*, which introduces logical *and*, *or*, and *not* operations on Boolean values. Then we have *predicate logic, set theory, and number theory*, often found in combination, which introduce sets, functions, and quantifiers *there exists* and *for all*, and numbers and operations on them. Finally we have *temporal logics*, which includes operators such as *at some point in the future*, and *at the next moment*.

6.2.1 Predicate logic

One of the best-known general-purpose logic languages used in computer science is *Z*. Z includes set theory and number theory, and several variants exist that extend it with special-purpose shorthands. In particular, we will look at *Object-Z*, which adds some object-oriented data structuring shorthands. Z has a special notation: a Z specification may be split into *schemas*. Each schema may have a title, and consists of definitions (at the top) and truth expressions (at the bottom), separated by a line. The truth expressions are correctness constraints, as a system can be said to satisfy the specification when these expressions are always true for that system. Z specifications are not executable, as are predicate logics in general (Wooldridge, 1998), since they are underspecified w.r.t. executable behaviour in the general case. The computational feedback possible is syntax checking, type checking, and proof assistance.

Other languages are the VDM (Vienna Development Method) specification language, and its computationally supported descendant PVS, which are similar in expressiveness to Z (Sheppard, 1995). Two languages that are similar but are more computationally tractable are Alloy and OCL (Object Constraint Language) (Dingel, 2002). Alloy enables automatic proof by limiting the context of proof to specific cases. OCL is part of the UML (Unified Modelling Language) and is an augmentation of UML structure diagrams. While the OCL standard is still under development, OCL is potentially the most well-defined language, as it is meant to enable assertion checking in any executable system that can be made to correspond to structure diagrams. Some implementations already exist that do this, notably, the Dresden compiler couples it to the Java programming language (Hussmann et al., 2000). Note that OCL assertion checking may be implemented by little more than regular programming language expressions, when the programming language has good support for sets, and some (rather straightforward) shorthands are provided. This brings us to one other kind of logic specification, which is the (very common) use of *assert* statements in programming languages, which enable correctness constraints to be expressed in certain points in the program, using just Boolean expressions in the programming language. Summarising:

Language	Computational Tractability	Well-definedness
Z, Object-Z	Syntax & type checking, proof assistance	full system
VDM, PVS	Syntax & type checking, proof assistance	full system
Alloy	Automatic proof with limited context	simplified system
OCL	Assertion checking	full system
Assert statements	Assertion checking	full system

The expressiveness of these languages are not quite the same, as the less computationally tractable variants allow the specification of meaningful yet computationally intractable properties, such as the transitive closure (i.e. the result of the recursive application) of an arbitrary relation (Geerts and Kuijpers, 2003).

Almost all of these languages have special facilities for specifying data structures. They are mostly similar to the tuple facilities provided by set theory. We may, for example, define a 5-element data structure as a tuple of 5 elements: $T = (a, b, c, d, e)$, with *a*, *b*, *c*, *d*, *e* the attributes or fields of the structure. The attributes of variable *t* of type *T* may then be addressed by means of a *select* operator (usually '.'): *t.a* is field *a* in variable *t*. This is very much analogous to the '.' operator found in popular programming languages. Some languages also have references, which are similar to pointers or references in programming languages. In OCL for example, a field may be a reference to another data structure, or a relation (which translates to a set of references). Selecting a relation therefore results in a set. In OCL, we may perform the select operator on sets, so that we may follow several relations in a chain. This is called *navigation*. In Z, we have more powerful facilities, such as the restriction ($\lhd$ and $\rhd$) and anti-restriction ($\lhd$ and $\rhd$) operators. Such usage of relations to determine sets is often similar in nature to database querying.

These languages specify properties that a system should have at particular moments in time. Evolution of the system over time is expressed in these languages only by specifying properties of the state of the system before and after a specific event occurs. One example are pre- and postconditions of operations. More complex temporal properties can only be properly described in temporal logic, see section 6.2.2.

These specifications are widely used in software specification, though assertion checking is by far the most popular. Much fewer have also advocated their use for interactive systems. There is still relatively little experience with using logic specifications to specify usability aspects inherent to interactive systems.

One example is found in (Wright et al., 1997), which proposes that logic specifications may form a suitable complement to the not-so-formal empirical evaluation methods that are based on concrete, simplified prototypes or scenarios. They propose a methodology in which requirements are formally specified and design options apparent from formal specification are worked out into scenarios that can be used for evaluation methods such as cognitive walkthrough. A preliminary experiment is done which consists of applying this method for several aspects of hypertext interaction, which indicates that the formal models do expose some usability issues, though the method is not without overhead. Further experience would be needed to form further conclusions.

Also relevant is the work by Hussey and Carrington (Hussey and Carrington, 1997a;

Hussey and Carrington, 1996b; Hussey and Carrington, 1997b), which concerns specification of GUIs by means of Object-Z. They advocate a separation of specification and design, with the specification being achieved by a declarative language such as Z. They also advocate the use of interactors, with each interactor being specified in a Z schema. They discuss several larger examples. Unfortunately, their empirical study on comprehension of these Object-Z requirement specifications versus natural language specifications did not show significant differences between these two kinds of specifications, except that the Object-Z specifications took longer to understand.

More work by Hussey et al. (Hussey et al., 2001) illustrates a GUI example in Object-Z which is evaluated using metrical evaluation of the specification with respect to some given use cases. However, their metrics have to be calculated by hand, rather than automatically, and it is not quite clear how to automate this process.

Example. Because we did not find any example of a logic specification of a VE, we will give one here. We will give an example of a Object-Z specification of a spatial first-person VE through which agents can navigate and communicate. Emphasis lies on how agents present themselves, and how they can move around and observe others, and what they are assumed to know of each other. Such a model may for example form the basis for a multimodal communication model, as it makes clear the space of possibilities of communication. In particular, it shows what an agent theoretically could and could not know. We will paraphrase and explain the specifications by means of natural language specifications.

This example was adapted from an early version of the same world, which was specified without any computational feedback. This version, however, contained errors that were found after further scrutiny. The final version was neatened, and was syntax- and type-checked using the *wizard* type checking tool. This was a significant help, as the tool uncovered various errors quickly during the re-specification process.

A VE consists of a space (a set of possible locations) inhabited by agents. Agents have avatars, which have a characteristic appearance to other agents. Agents have pointers (like mouse pointers) which they can point at objects in the VE, and which other agents can see. The agents' view may be a first-person perspective, so that the visibility of each agent depends on its position (a value of type POS) and its orientation (a value of type ANGLE). First, we define a few types:

$[POS]$ *An avatar′s position*
$[ANGLE]$ *An avatar′s orientation*
$[APPEARANCE]$ *An avatar′s characteristic appearance*
$[UTTERANCE]$ *A* $(natural\ language)$ *utterance*
$[ID]$ *The agent′s unique ID value*

An Avatar is an agent's embodiment. This consists of its appearance, and its position and orientation in the environment. Its Z schema is just a data structure with three elements:

Avatar

$$appearance : APPEARANCE;$$
$$angle : ANGLE;$$
$$pos : POS$$

A Pointer is the state of the agent's pointer. Again its schema is a data structure:

Pointer

$$appearance : APPEARANCE;$$
$$pos : POS$$

Speech is the information conveyed by an agent's utterance. The utterance is somehow characteristic, so that the agent that speaks can be identified. This is modelled by the speech having an APPEARANCE value.

Speech

$$appearance : APPEARANCE;$$
$$utterance : UTTERANCE$$

View is what the agent is seeing through its eyes, in other words, what a user would see on his/her screen.

View

$$avatars : \mathbb{P}\, Avatar;$$
$$pointers : \mathbb{P}\, Pointer;$$
$$speech : \mathbb{P}\, Speech$$

Now, we can define an Agent. Note that INIT defines its initial state.

Agent

$$
\begin{array}{l}
id : ID; \\
avatar : Avatar; \\
pointer : Pointer; \\
speech : \mathbb{P}\, Speech; \\
observing : View; \\
knowledge : \mathrm{seq}\, View \\
\hline
last\, knowledge = observing \\
\#speech \leq 1
\end{array}
$$

INIT

$$
knowledge = \varnothing \times \varnothing
$$

This is the first schema that is more than a simple data structure. Note that *speech* is a set of elements of type *Speech* (i.e. an element of the powerset $\mathbb{P}\, Speech$) and *knowledge* is a sequence (i.e. an ordered list) of *View*s. The two lines of constraints specify that *observing* is part of *knowledge* and is in fact found at the end of the sequence (indicated by *last*), and that *speech* is a set of zero or one elements (i.e. the agents says something or is quiet).

In Z, it is common practice to define the state of a system by means of a definition containing all the variables in the state, an initial state, and a state change. We start with the state definition of our virtual world:

WorldState

$$
\begin{array}{l}
space : \mathbb{P}\, POS; \\
agents : \mathbb{P}\, Agent; \\
visible : POS \rightarrow ANGLE \rightarrow \mathbb{P}\, POS; \\
hearable : POS \rightarrow \mathbb{P}\, POS; \\
moveable : POS \leftrightarrow POS \\
\hline
\forall\, a : Agent \bullet a.speech \subseteq a.observing.speech \\
\forall\, a : Agent \bullet a.pointer \in a.observing.pointers \\
\qquad \land\, a.pointer.pos \in visible\ a.avatar.pos\ a.avatar.angle \\
\forall\, a, b : Agent \bullet b.avatar \in a.observing.avatars \\
\qquad \Rightarrow b.avatar.pos \in visible\ a.avatar.pos\ a.avatar.angle \\
\forall\, a, b : Agent \bullet b.pointer \in a.observing.pointers \\
\qquad \Rightarrow b.pointer.pos \in visible\ a.avatar.pos\ a.avatar.angle \\
\forall\, a, b : Agent \bullet b.speech \subseteq a.observing.speech \\
\qquad \Rightarrow b.avatar.pos \in hearable\ a.avatar.pos \\
\forall\, a, b : Agent \bullet (b.avatar.pos, a.avatar.pos) \in moveable^{+} \\
\mathrm{ran}\, moveable \subseteq space
\end{array}
$$

These constraints are not very easy to read without a small explanation of their intent. The first two lines specify that an agent's speech and pointer can be heard and seen by the agent itself. The next three lines specify the constraint that an agent views only those things that are within its Visible and Hearable range. The second last line specifies that each agent is reachable by all other agents (with reachability being specified by the transitive closure of moveable, $moveable^+$). The last line specifies that an agent can never move out of the VE space.

We now specify the initial state of the world. We only specify that the world begins with no agents, and that there are no constraints on the initial state of the other variables:

Init

$$WorldState'$$

$$agents' = \varnothing$$

Now, we can define what happens when something changes. In our case, time proceeds in time steps and agents may act in parallel in each step. In this schema, $\Delta WorldState$ stands for the WorldState before and after a time step. The variables after a time step are denoted by a tick ($'$) symbol (i.e. $agents'$ is *agents* after the time step).

TimeStep

$$\Delta WorldState$$

$$visible' = visible$$
$$hearable' = hearable$$
$$moveable' = moveable$$
$$space' = space$$
$$\forall a, b : Agent \bullet a \in Agents' \land b \in Agents \land a.id = b.id \Rightarrow (a.avatar.pos, b.avatar.pos) \in moveable$$
$$\forall a, b : Agent \bullet a \in Agents' \land b \in Agents \land a.id = b.id \Rightarrow \#a.knowledge = \#b.knowledge + 1$$

With the first four lines we specify that the Space and the three agent functions are constant over time. With the second last line, we specify that an agent can only move within the range of the Moveable function, and that one item (the last Observation) is appended to its Knowledge list.

While we do not do much with the agent's Knowledge, it may be used to specify what is possible for an agent to know and what is not, which may form a basis for communication behaviour. For example, if an agent speaks, it may not refer to something that is not in Knowledge.

Discussion. While specifying, one often finds oneself trying to specify an intuitive property that translates to several logical constraints that, together, specify the property. An example of this is the five constraints in WorldState that specify the contents of the Observing variable. One problem with such constraints in general is that it appears

difficult to verify if a set of expressions are correct and (reasonably) complete with respect to desirable behaviour. When reading the expressions, it may be of great help to annotate them with the property they are trying to describe, so that the intentions of the designer are made clear. Reading them is in fact so difficult, that some refer to alternative specifications. If we look at the examples in the previously-discussed paper (Wright et al., 1997) for example, the Z specifications of document navigation specified there are made more understandable by means of structure diagrams, illustrating the document structure. That brings up the question: what added value is the Z specification compared with these diagrams? In this case, there was no real correctness issue, and it is likely that the diagrams used were a more appropriate specification language than Z.

Experimental results on the usefulness of logic languages are not conclusive. The work of Roast et al. on using Z on a simple (interactive) software problem by beginners (Khazaei and Roast, 2001; Roast and Siddiqi, 1999) shows that Z may have a *negative* impact on solution quality. They conclude that people who are not very experienced with the language will be hindered to some extent by the difficulties of the language, and that they may be forced to produce the simplest rather than the most generic solutions to overcome the difficulties of specification. This goes against the principle of declarative specifications to enable specification of generic solutions to preclude early commitment to implementation details. We might conclude that, at the least, logic languages are not well suitable for HCI specialists and user centred development, which makes its usage as UI specification language more problematic.

While type checking proved helpful, more computational feedback would be preferable. This could be achieved by translating the specifications to other languages, such as OCL or Alloy. There are several problems that occur when trying to do this. Firstly, we may note that some of the assertions are computationally intractable. In particular the functions *Visible*, *Hearable* and *Moveable* are abstract functions that may have any domain and range. This makes some of the assertions intractable, in particular the reachability property $\mathit{forall}\ a, b : \mathit{Agent} \bullet (b.\mathit{Avatar}.\mathit{Pos}, a.\mathit{Avatar}.\mathit{Pos}) \in \mathit{Moveable}^+$ implies computing the transitive closure of the *Moveable* function. While reachability is a quite meaningful property, it cannot actually be verified directly. Secondly, if we want to convert the specifications to assertion checks, we should determine what these functions stand for. *Hearable* may be a straightforward piece of code, such as a simple and explicit distance function, that could be used in assertion checks. The domain and range of the function could be made explicit, so that some properties of it may be checked. *Visible* and *Moveable* are more problematic. For a user, these may be implicit functions of a rendering engine component, and their domain and range may not be known. It should be clear from this that the reachability property of the rendering engine is quite impossible to check, at least not without creating a non-straightforward mapping between specification and implementation (such as an abstraction step). Note that *Knowledge* may, but need not, directly describe a variable in the program, since we have stated that it indicates what an agent *might* know.

This small assessment indicates that computational tractability and well-definedness are real issues here, that cannot be solved by a simple automatic translation into another language, but may require an interpretation step. If we follow incremental methodolo-

gies (such as Agile methodologies), we may be tempted to give priority to assertion checking possibilities to ensure proper consistency between program code and specifications, but must do this at the expense of losing expressiveness, and produce incomplete requirement specifications. This may in practice be acceptable, though.

Conclusion. Our preliminary conclusion is that logic models are likely to be useful for specifying user interfaces, but it is also likely they will not be adequate for usability analysis, in particular because one needs to be well trained in logic to use it effectively. As regards specification of *usability requirements*, further research is needed. Assertion checking approaches are the most well-used, and have the greatest potential.

6.2.2 Temporal and intentional logics

While there are several different logical approaches for describing time, what they share is the ability to express statements about the past or future, and on the temporal ordering of events. We will concentrate here on *temporal logic*, which is the most common. Still, it is much less common than predicate logic. Temporal logic requires some model of multiple possible worlds, or history and future, so, when it is combined with predicate logic to achieve a reasonable level of expressiveness, the result is something rather complex. Possibly, it is this added complexity that precludes its wide usage.

Temporal logic only has a couple of basic operators:

$\Box x$: from now on, x holds
$\Diamond x$: eventually, x will hold

Various others exist for specifying properties about history and future, but these can typically be derived from these two.

If we look at existing languages and tools for temporal logics, there are a few that are worth mentioning for their computational tractability. One is Metatem, which is an executable temporal logic. To make it executable, expressions are rather limited however, and predicate logic expressions are not allowed. The language is suitable for simplified models only. Another is the use of similar temporal logic expressions in automatic verification, as found in some state automata based languages (see section 6.4). Examples of this are Promela, and some HCI verification language(s) (Mezzanotte and Paternó, 1996). Here, the logic only applies to simplified models, rather than full systems. No temporal logic language exists that can be used for assertion checking in a language that produces a full executable system (Wooldridge, 1998).

Another class of logics that has attracted some attention in the context of agents are intentional logics. Intentional logics are general logical frameworks for describing agents' mental states in high-level terms, such as goals, intentions, and beliefs. In fact, some argue that such explicit modelling of intentionality is what truly characterises an agent system (Meyer, 1998). Intentional logic specifies high-level psychological concepts such as desire and intention. So, it requires temporal logic as a basis. Typically, intentional logic frameworks define extra sets of operators on top of some existing variant of temporal logic. Examples are *Belief Desire Intention* (BDI) (Meyer, 1998), based

on temporal logic, and *Knowledge Abilities Results Opportunities* (KARO), based on dynamic logic. To give an idea of these logics, here are some operators from BDI (Veer et al., 1998, the sheets by J. J. Meyer):

$K_a x$: a knows that x holds
$B_a x$: a believes that x holds
$G_a x$: a has a private goal to achieve x
$I_a x$: a intends to achieve x

Using these operators, we might for example specify the rule:

$B_a I_b x \wedge B_a \neg \exists_{c \neq b} B_b G_c x \rightarrow B_a G_b x$

If a believes that b intends to achieve x, and it believes that b does not believe that there exists some other agent that wishes to achieve x, it will assume that b also has x as its private goal. In other words, if b doesn't seem to do x for some other agent, it probably does it for its own private purposes.

Intentional logics may be used for tracking the intentions of other agents during communication, or task delegation in team formation (Dignum et al., 1999). The idea is that an intelligent agent can be specified or designed by specifying its knowledge and reasoning using such rules (but preferably, simpler ones from which other rules, such as the rule above, may be inferred). However, there are serious suitability problems with intentional logics (see also (Müller, 1996, section 2.3.3)):

- It is not easy to write down something meaningful using these kinds of 'advanced' logic. Even in specifying small systems, experts are known to make serious mistakes. A particularly infamous example is the Little Nell problem (Veer et al., 1998, J. Meyer's lecture). Other examples are the ill-definedness of some of the intentional frameworks that have been proposed (Baltag, 1999), and of the popular agent communication standards (van Eijk et al., 1999).
- While there is obvious need for computational feedback, we have already seen that neither predicate logic nor temporal logic are computationally tractable. The only computational feedback tools that remain are theorem provers and syntax checkers.
- The relation between intentional logic and computer programs are not even well-defined, as pointed out in (Wooldridge, 1998).

Conclusion. Temporal logics have some potential usefulness in checking simplified models of a system, which means that such a simplified model should be available at some point in the development process. Intentional logics come out even less favourable: while they certainly are expressive, in the light of the other three requirements, their usefulness as specification languages is questionable.

6.2.3 Production rule systems and Prolog

Some executable logic-related languages exist that have been used extensively for various applications. Many (though not all) of the expressions in these languages are

very close to logical expressions. Particularly well-known is *Prolog (PROgramming in LOGic)*, which has been used for many things, but especially for programming agent behaviour. Examples of this are the mVITAL multi-agent VE system (Anastassakis et al., 2001), and DESIRE (Jonker and Treur, 1998a). A less usual example of the application of Prolog is found in the multi-user access policy language COCA (Li and Muntz, 1998).

Besides Prolog, there is also a class of languages called *production rule languages*, which have a similar form. Examples are OPS-5 and Soar. A simpler kind of production rule system is also found in *active databases*. Production rule languages are often coupled with human simulation models, in which they play the role of describing a human's or interface agent's tasks and knowledge. Examples of this are the Steve and Adele systems (Shaw et al., 1999; Rickel and Johnson, 2000). Another important application of production rules is user task modelling and user simulation. A clear example of this is the Cognitive Complexity Theory (CCT) task language (Haan et al., 1991). Production-rule-like specifications are sometimes used for NL dialogue, as in the Speechmania system (Philips, 1998). There are also agent architectures that use logic-based languages, such as InterRAP (Müller, 1996).

While there are various differences, the general form of specifications of Prolog and different production rule languages are similar, and production rule systems may be naturally implemented using Prolog specifications. Both distinguish facts, and rules, with a left-hand side (the condition) and a right-hand side (the action or consequence). Some production rule systems are forward chaining (they just apply any applicable rules to the facts, creating intermediate results, one rule at a time), or backward chaining (tring to find a particular value by finding a sequence of rule applications that determines the value). Prolog is closest to a backward chaining production rule system, but executing rules in Prolog does not change any facts, but just produces intermediate results which are forgotten after the goal is found.

In its most basic form, production rule specification is the complement of state automata (see section 6.4) specification: the ordering of actions is not described explicitly. Instead, the state of the system is described as a number of facts, and the behaviour as production rules. Each rule condition is an expression in terms of facts, and each rule action may change some facts. If several conditions hold simultaneously, one may be chosen, perhaps according to some other criterium. Here is a small example (from an expert system) (Pachet, 1995):

```
decideNoTreatment
     "If pressure is normal then consider abandoning
      treatment"
     |Doctor d. Patient p|
             p bloodPressure < d maxBloodPressureFor: p.
             p hasTreatmentForTension.
     actions
             d considerAbandoningTreatmentFor: p.
```

Note that if the condition in a condition-action rule is not made false by the action, the

rule will be triggered endlessly.

Goals and plans. A production rule system often needs several actions to achieve a certain goal. During the performance of the actions, facts may change in unexpected ways, by other (external) rules, possibly changing the course of actions to be taken, or initiating new actions. This means that the system's reaction to some conditions needs to be split into a chain of actions. This is called *control structure* (Cooper and Wogrin, 1988), analogous to control flow in procedural languages. This is usually done by introducing extra variables named *task* or *goal*, which stand for pending goals, and determine in which phase of execution the system is. Here is an example in CCT (taken from (Haan et al., 1991)):

```
(PDELW1 IF (AND      (TEST-GOAL delete-word)
                 (NOT (TEST-GOAL
                            move cursor to %UT-HP %UT-VP))
                 (NOT (TEST-CURSOR %UT-HP %UT-VP)))
        THEN ((ADD-GOAL move cursor to %UT-HP %UT-VP))
)
(PDELW2 IF (AND (TESTGOAL deleteword)
                 (TESTCURSOR %UTHP %UTVP))
        THEN ((DOKEYSTROKE DEL)
              (DOKEYSTROKE SPACE)
              (DOKEYSTROKE ENTER)
              (WAIT)
              (DELETEGOAL deleteword)
              (UNBIND %UTHP %UTVP))
)
```

These two rules define how a user may delete a word in a text editor. The first rule defines that the cursor has to be placed on top of the word in case this was not yet the case. The second rule defines how a word below the cursor may be deleted. Unlike structured programs and plain hierarchical task models, these models enable arbitrary manipulation of the goal state. In this example, this is done by means of `ADD-GOAL` and `DELETE-GOAL`.

Pattern matching. Many production rule systems have some kind of pattern matching facility built in. In fact, it is already found in the CCT example above: in the first rule, the actual cursor position is matched with the goal cursor position. The following example 'virtual environment' agent behaviour specification from OPS-5 (Cooper and Wogrin, 1988) illustrates more clearly how it works:

```
(literalize monkey  at on holds)
(literalize object  name at weight on)
(literalize goal    status type object to)
...
```

```
(p mb7  (goal ^status active ^type holds ^object <w>)
        (object ^name <w> ^at <p> ^on floor)
        (monkey ^at <p> ^holds nil)
    -->
        (write (crlf) grab <w>)
        (modify 3 ^holds <w>)
        (modify 1 ^status satisfied))
...
```

The three `literalize` statements define three tuple data types `monkey`, `object`, and `goal`, with, respectively, length 3, 4, and 4. Any number of tuples may exist of each type. The set of all tuples comprise the set of facts of the system. Pattern matching consists of matching specific fields, usually for equality, across facts that are tested for in the condition part of a rule. If a match exists, the rule is triggered and the values of the fields of the match found may be used in the action part. In this example, the rule `mb7` specifies how a monkey picks up an object: if it has a goal to pick up some object `<w>`, and that object `<w>` happens to be in the same location `<p>` as the monkey, it is picked up, and the goal is marked as satisfied.

Beside agent behaviour in a VE, it is also useful for contextual help systems, as the following (pseudocode) example from (Moriyon et al., 1994) illustrates. The identifiers starting with question marks are the fields to be matched.

```
When    ?object is selected for help
    and ?object has an interaction technique ?inter that
        invokes command ?cmd
    and command ?cmd has an input ?inp
    and interaction technique ?inter sets input ?inp
        to value ?val
then display the interaction message
  ...
```

Pattern matching may be viewed as the ability to use 'SQL' type database queries with 'SQL' join operations in the condition part of a rule. The fact set has in fact the same form as found in SQL type databases. This means that pattern matching may be done using a regular database query engine.

Interaction. An agent's interaction with the outside world may be modelled as facts that change spontaneously. Some models exist specifically for building multi-agent systems using production rules, which illustrate how such specifications can be used in a larger, interactive system. A particularly interesting one is DESIRE (Jonker and Treur, 1998a; Brazier et al., 1995; Brazier et al., 1994). DESIRE is a hierarchical compositional model: each agent may in turn be composed of agents. The agents are connected by point-to-point channels, which may be disabled and enabled while the system runs. Each channel links the truth value of one component's fact to one of another component's fact. The mapping between these facts is changeable using a

translation table. Component-environment and component-human interaction can also be described by viewing resp. the environment or human as a separate component. Each agent may have one or several rule sets. Here is an example:

```
if     belief(at_position(food,P:POSITION),pos)
   and belief(at_position(screen, p0),neg)
   and belief(at_position(self,P:POSITION,neg)
then to_be_performed(goto(P:POSITION))

if     belief(at_position(food,P:POSITION),pos)
   and belief(at_position(self,P:POSITION),pos)
then to_be_performed(eat)
```

These rules specify the 'world interaction management' component of a mouse searching for food, given a fixed setup with a number of discrete locations and an impassable screen which may be in place or absent. The belief() facts are updated from another component, the 'world information maintenance' component, while the to_be_performed assertions are output again to other components. This example also illustrates the use of data types, which may be used to constrain pattern matching. `POSITION` is a data type specifying a specific subset of the set of facts. A data type may be part of a type hierarchy, which may be specified separately.

Production rule systems enable description of very complex behaviour, and have been successfully used in various AI applications. In fact, complex behaviour may be specified with only a few rules. However, systems with many rules easily run the risk of becoming unmaintainable. (Cooper and Wogrin, 1988) describes some of the development issues. One technique to increase modularity is the introduction of control structure to structure rules into independent clusters of rules (which is somewhat analogous to structured programming). Another way to introduce modularity is a scheme such as used in DESIRE, which separates the system into neatly separated modules of facts and rules. Two other problems mentioned in (Cooper and Wogrin, 1988) are the difficulty of producing requirement specifications, and analysing a production rule system's control flow, both of which make testing more difficult.

Production rules have been used extensively, and already have some successful history in HCI task modelling, showing that it is usable by other people than computer experts. On the other hand, Prolog has been known as difficult to use for beginners, as compared to procedural programming languages. A cognitive dimensions study by Green et al. (Green, 1999) suggests that lack of role-expressiveness (i.e. mapping of statements to the programmer's intentions) is an important factor in this difficulty, and might be solved by just syntactical improvements of the language.

Conclusion. Production rules or Prolog may be interesting for programming agent behaviour. They may be integrated into a bigger system using a method like that found in DESIRE. There are, however, various approaches which need to be examined more closely. While we propose the use of such languages as a topic of future research, our current framework does not support them, as they are not deemed essential for the

basic framework which emphasises system architecture. Production rule based agents are likely to fit into the model as components.

6.3 Structural models

Structural models are models that enable the designer to structure a specification to improve readability or reduce redundancy. In part, structure is independent of specification semantics or program behaviour, as the same behaviour may be structured in many ways. For example, program functions may be transferred from one module to another. Changing structure without changing behaviour is sometimes called *refactoring*. Some tools are now available to automate the refactoring process.

There are different ways to structure software. We will distinguish two classes of structuring: data structures and dataflows. Both are well known in SE, and both have variations and families of languages that are based on them. Structured notations often come with diagrammatic notations: we have data structure diagrams and dataflow diagrams. Such diagrams have the potential of being used by non-software experts (Lim and Long, 1993).

6.3.1 Structure diagrams

Structure diagrams have a long history and are well-used. The most basic structural specifications are abstract, diagrammatic notations that are used to supplement program code in a variety of programming languages. They fit especially well with object-oriented languages and database languages (such as SQL (Date, 1997)). Examples are *entity-relationship diagrams* (ERDs) and *UML object diagrams* (Green and Benyon, 1996; Object Management Group, 2001). Both are similar in notation, and the more modern UML diagrams have the same expressiveness as well-known variants of ERDs.

While structure diagrams are abstract and hence not executable, various (CASE) tools now exist that couple such specifications to executable code, which is mostly done for object-oriented languages. For example, coupling is achieved by generation of code skeletons (Pinheiro da Silva, 2000), or extracting diagrams from code, or active consistency maintenance between diagrams and code.

Usually, structure diagrams are used to describe the application domain data, or the coarse structure of the application as a whole. There are some exceptions to this, though. One of the most notable is the *E/R Modelling of Information Artifacts* (ERMIA) (Green and Benyon, 1996; Green and Benyon, 1995) method. This method is a usability method that models the structure of user interfaces and other interactive devices to a fine level of detail. ERMIA specifications are meant to be suitable for non-programmers. They are typically not coupled to executable code, but they might be. ERMIA has been used for GUIs and Web pages. Other interesting exceptions are OOHDM and RMM (Rossi et al., 1999; Isakowitz et al., 1995), which prescribe a structural notation of Web pages,similar to structure diagrams.

ERDs may also be used to specify the architecture of user interfaces, as specified ac-

cording to a specific architectural style. This provides an abstract graphical notation of an application's architecture. Even using plain ERD specifications in this manner is likely to provides some added value already, as such architectures are sometimes supplemented by diagrammatic notations, which may be better specified using ERDs. For example, if we consider the PAC model (see section 4.2.1), we may observe that typical PAC tree diagrams only sketch the structure of a user interface at a specific moment in time. If we simply represent tree nodes by entities and arcs by relationships, we get an ERD representation. The relationships need not even be given names, since they are all of the same kind. With this representation, we are already able to specify the general pattern that the tree should conform to at any moment in time, by using relational multiplicities. This kind of diagram concisely specifies the structural correspondence of the agents in the system. In fact, such a representation is similar to one that might have been produced by an ERMIA specification.

As an example, consider the PAC model for a simple multi-user editor, which consists of editing windows and an overview window showing which users are editing particular files. An ERD specifying the general structure of the system is specified in figure 6.2. It specifies that there are only two kinds of windows, editor windows and overview windows. At any moment in time, a user may have zero or more editor windows open, and may or may not be viewing the overview window. The overview window contains zero or more file views.

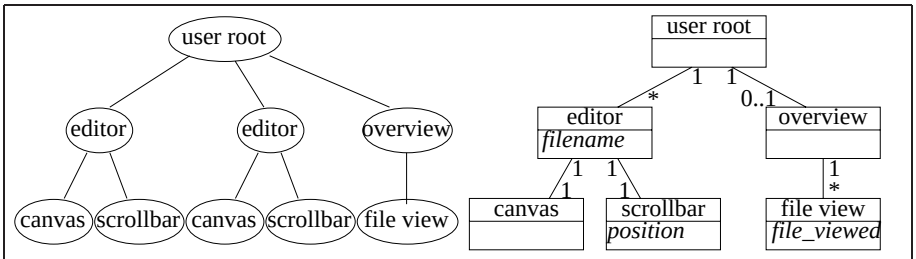
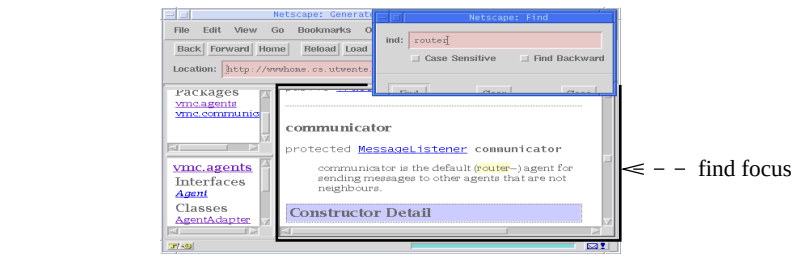


Figure 6.2: PAC architecture of multi-user editor and its ERD.
Left is an informal diagram illustrating the structure of one user's instance of a multi-user editor at a particular moment in time. The user has two editing windows open and one overview window with one file view. Right is an ERD specifying the general pattern that the editor's component structure may conform to.
A box indicates an entity, with the entity's name at the top, and its fields at the bottom. A line indicates a relation. Multiplicities are denoted by numbers or ranges, with '*' meaning zero or more. Note the attributes *filename*, *position*, and *file_viewed*, which correspond to user interface state required for the file views.

ERDs may effectively be used to clarify design choices in the multiplicity of graphical elements in GUIs, the correspondence between elements, and the visibility of this correspondence. As an example of this, consider the use of the Find box (the dialogue box

that is used to search in a web page) in the Netscape browser. People who have used Netscape extensively may know that the Find box may remain open indefinitely, but that keeping it open may result in confusion. In fact, Netscape has multiple Find boxes, one for each window. It is by no means clear which box corresponds to which window, which may be considered a serious usability problem, as Find boxes sometimes get lost behind other windows. At the same time, the Find operation operates on one specific frame, namely the frame that has the keyboard focus. We specified this design feature, along with two alternatives that came to mind easily, in figure 6.3.



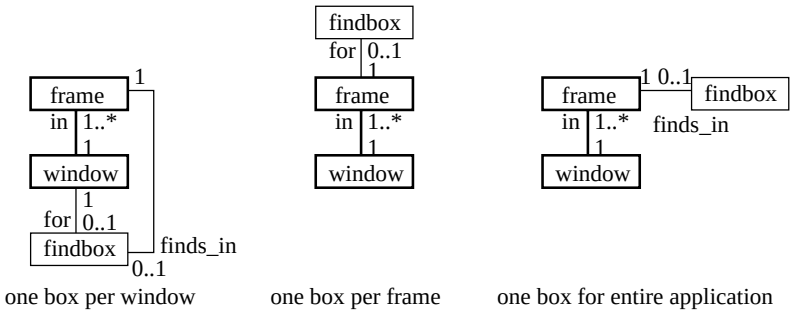


Figure 6.3: ERD modelling the Netscape find box.

Conclusion. Structure diagrams show potential for modelling user interfaces, in particular GUIS and Web pages, and possibly VEs as well. Their diagrammatic notation and high abstractness makes them interesting for usability purposes. They have as yet not been used extensively for these purposes, and the existing attempts are still lacking in computational feedback facilities, even though computational tractability and well-definedness of structure diagrams is high. They are interesting to use as a basis of VE modelling, and the specification technique we propose in part III is data structure based, enabling structure diagrams to merge naturally with the program code.

6.3.2 Dataflows, configuration languages, and glue languages

Dataflow-based specifications are specifications that are centred around the identification of software modules or components and the data that is transferred between them. The simplest kind of dataflow-based specification is the *dataflow diagram* (DFD). This is a directed graph, with the nodes being the components and the arcs indicating the existence of data transfers between pairs of components. We consider a specification dataflow-based when nodes and arcs are explicitly identified, and are specified in a single concise specification. Dataflow-based specifications could be generated naturally from typical modular programming languages, with the modules forming dataflow nodes, or from other specifications in which nodes can be identified.

We will identify three classes of dataflow-oriented languages: *dataflows*, which have fixed structures, like dataflow diagrams, *configuration languages*, which identify nodes and arcs, but enable the dynamic creation and deletion of them, and *glue languages*, which identifies nodes, but connects them using arbitrary 'glue specifications', rather than just point-to-point arcs.

Dataflows. Dataflows are useful for graphical systems, and may effectively specify data dependencies between components. They can be used to specify reaction to updates, that is, propagating the effects of a value change occurring within a process through the system towards the relevant processes. They can also be used to process continuous streams of data, which is for example useful to produce animation. Dataflows are most effectively specified in a graphical syntax, and various software tools enable programming by direct graphical drawing of dataflows. They have been used to specify visualisation and animation, such as the IRIS Explorer system, in which nodes process continuous data to form graphical output (Halse, 1991). A number of node types are available from a library, which may be configured by setting parameters. IRIS nodes may include interactive user interface objects, such as windows in which parameters can be changed during the visualisation.

(Banavar et al., 1998) also describes a graphical dataflow editor for specifying multi-user applications. The nodes in the dataflow are user interface objects, such as text fields and buttons, or nodes for managing communication.

Another example of dataflow specification is the Marigold system, which specifies the behaviour of VE objects using a combination of dataflows with Petri nets (see section 6.4.3). It provides a graphical editor, and produce runnable code through integration with a 3D graphics system.

Another interesting example is the VE specification language VRML (Carey and Bell, 1997). A VRML world consists of nodes, which contain data and mostly represent various types of basic graphical objects. Most nodes have specific input and output points. An event arriving at an input point triggers a procedure inside the node, which may change the node's data, and produce events through one or more of the output points. The most basic example is a node storing a data value: it has one input point for changing its value, and one output point for notifying its value changes to other nodes. There are also output points that spontaneously generate events, such as timer and

user-triggered events. Arbitrary routes connecting input points to output points may be specified separately, by means of `ROUTE` commands, as long as each input's data type matches each output's. Each input/output point may have several routes leading to/from it.

Beside dataflows, other kinds of behaviour are determined implicitly according to the node hierarchy. In VRML, all nodes (including physical objects) are conceptually arranged in a hierarchy, which represents an aggregation hierarchy. Some nodes determine the position and angle of their child nodes, so that groups of objects may be rotated and translated by a single operation. Sensors may be declared for sensing user click and drag operations on groups of nodes. Each sensor enables and detects operations done on its sibling nodes, and on any nodes further down the hierarchy. A sensor's sphere of influence may be overridden by another sensor situated further down the hierarchy.

Like other dataflow models, VRML's route (and node) structure is fixed in the specification. Both routes and nodes can be added and deleted, but only by means of functions called inside scripts, which are written in other programming languages.

Configuration languages. We identify *configuration languages* as languages that enable the maintenance of dataflow graphs by creating and deleting nodes and arcs. Interactors and architectural styles are mostly dataflow-oriented, although changes in the user interface structure, and hence in the graph structure, are common, so they cannot be described fully using just dataflows. However, they could well be described using a configuration language. While typical interactor models, such as PAC and MVC, only loosely describe guidelines, combination with a configuration language may increase concreteness and adherence to the guidelines.

In the SE literature, there has been increased attention to such languages, which are also called *architecture description languages* (ADLs) there. Several overviews and classifications of such languages have been given (Medvidovic and Taylor, 1997; Issarny et al., 2000). The most characteristic property of these languages is the existence of components (nodes) and connectors (arcs) as basic building blocks. There are typically different types of connectors (such as synchronous, asynchronous, publish-subscribe, etc.) as well as components. Some connectors enable multiple components to be connected. These languages are also interesting in that they enable distributed system issues to be addressed using these different types of connector. However, while we are interested in languages that enable dynamic creation and deletion of nodes and arcs, only a subset of these languages do support this.

Instead of using such specialised languages, dataflow-oriented specifications may also be written in regular programming languages, with the loss of some conciseness and uniformity. In fact, we find dataflow-oriented constructs in Java, such as the `addListener` and `addObserver` methods, which create dataflow arcs between pairs of objects. Here is a typical example, describing a minimal application using the MVC model, written in Java:

```
//// instantiate nodes
```

```
  Button setbutton = new Button("Set");
  Button clearbutton = new Button("Clear");
  model = new MessageModel();
  view = new MessageView();
//// note: implicit arc is created here by passing
////       a reference
  controller = new MessageControl(model);
//// connect the nodes
  setbut.addActionListener(controller);
  clearbut.addActionListener(controller);
  model.addObserver(view);
```

This application creates two buttons that can be used to set and clear a message string in the model component, through a controller component. A view component shows the string, and keeps track of changes, which are notified by the model to the view. The arc specifications are given explicitly by the addActionListener and addObserver methods, which are found in the standard Java class libraries. So, the specification is already largely dataflow-oriented. However, extracting the dataflow-related part from a Java specification is not easy, as there is no uniform method to do this. There are other ways to communicate, so that an explicit dataflow specification of all communication is not enforced. In our example, the controller is created by passing a reference to its model, so that it accesses it using a regular method call. Still, since program code can produce dataflow-like specifications, architectural styles are often available as software libraries rather than programming languages.

A particularly interesting ADL is Chiron-2 (Taylor et al., 1996), because it is one of the most well known, and is one of the few that has been used for describing GUIs and animation. Actually, Chiron-2 is an architectural style that is available both as a software library and in the form of an ADL. Chiron-2 connects components through buses, which connects a number of components requesting services to a number of components providing these services.

Glue languages. We classify specifications that 'glue' components by other means than connectors or dataflow routes as *glue specifications* (which is analogous to the common term 'glue code'), which may be specified in specialised languages which we call *glue languages*. In SE, this kind of specification is also called *coordination language* or *middleware*. Unlike dataflows, glue languages may specify communication without explicitly specifying source and destination, which may be less simple to understand, but may provide the extra flexibility that is desirable for systems with a dynamical number of users or components. In SE, this is called *decoupling in space* (Kielmann, 1996), which takes various forms, with the simplest being shared dataspaces (Bjornson, 1992) or publish-subscribe services, and more complex forms such as *agent facilitation services*, in which agents can search for other agents providing specific facilities (Labrou and Finin, 1997). Various glue languages have been proposed for groupware, as well as for GUIs and user interfaces with graphical animation. We will discuss a few of the most interesting here.

A glue language that implements the parallel execution model is Reactive-C (Boussinot et al., 1998), and its more recent Java implementation, SugarCubes. In this system, each node may be connected to one 'sync' node, denoting a synchronous area. All nodes connected to the sync node execute in a parallel fashion, using a discrete-time model. Events may be sent to the sync node, which will multi-cast the event simultaneously to all other connected nodes. Each of these nodes may have chosen to subscribe to specific types of event only, and will discard any other types of event. A node may also send an asynchronous message to any sync node. In this case, the delay between sending and delivery may be arbitrary. See figure 6.4. Oddly, while the parallel model is found often in computer games, Reactive-C is one of the few that formalises it into a general technique. Typical Reactive-C examples include GUI-like applications with animation and collision detection and handling.

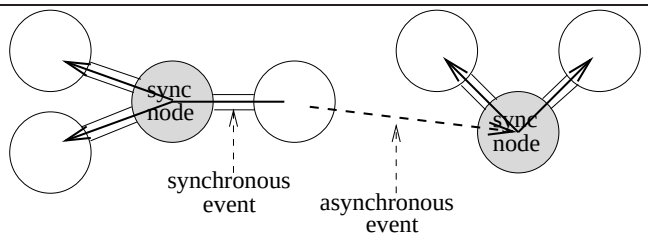


Figure 6.4: Reactive-C event model
The figure shows two sync nodes with nodes attached, indicating two synchronous areas. Synchronous messages (thick line) may sent to a sync node, which sends it to the other nodes within the same time instant. Asynchronous messages (dotted thick line) may be sent to other areas, but may have an arbitrary delay.

There are various languages found in the groupware literature. One example is DCWPL (Cortés and Mishra, 1996), which makes a distinction between *agents*, such as users, who may issue actions of their own accord, and *artifacts*, which only react to actions, though the reaction may consist of the immediate issuing of more actions. Each agent is simply assigned a role name, which determines how artifacts react to actions from that agent. The artifact specifications describe how concurrent requests on components are scheduled. For each type of artifact, the conditions and procedures for creation, deletion, and function call actions done on instances of that artifact may be specified. Finally, policy specifications provide shorthands for often-occurring scheduling schemes. From a compositional viewpoint, DCWPL is not ideal, as neither the artifact function call specifications nor the information related to constraints between artifacts are compositionally structured. Instead, they have a global scope.

COCA (Li and Muntz, 1998) provides a similar model, but uses a Prolog-like language to integrate behaviour that provides more powerful language constructs than DCWPL. In COCA, the main language construct is the *role*. It is similar to the DCWPL artifact construct. Each role specification may specify local communication channels which connect each instance of the role to a regular software component running on a specific

machine. Still, like DCWPL, the users are connected by means of one global channel, and constraint information is global to all roles.

Conclusion. Dataflow-oriented specifications may be used for describing the architecture of GUIs, architecture and communication in groupware, and the dynamics of complex animations. There are many variations though, each with advantages and disadvantages.

Of the three dataflow-oriented variations, dataflows are obviously the most tractable, but have the disadvantage of not being able to specify changes in structure easily, which is essential for more complex systems. It is clear, though, that they have a function in describing parts of a system which themselves have a relatively fixed structure, for example, animated objects. While we do not provide a dataflow model in our current specification technique, we will describe how a dataflow system for describing animation, such as VRML, might be integrated with it.

Configuration and glue language approaches are more interesting for describing the overall architecture of the system. Because VEs, and especially multi-user and multi-agent VEs have a dynamical structure, it is desirable to provide some level of decoupling in space, which may effectively be achieved by a glue language approach, which is the approach we will take in our specification technique.

6.3.3 Constraint programming

Constraint programming is programming by specifying interdependencies between variables by means of a set of equations. When one variable is set or changed, the values of the other variables are determined by determining the solutions to the equations. One distinguishes two kinds of constraint systems: one-way and multi-way. One-way constraint systems do not actually need to solve equations, as they only specify the values of variables as functions of others. They are in fact similar to dataflows or spreadsheets, in which each value may be specified as a function of other variables. In multi-way constraint systems, one may specify sets of equations that will be solved automatically by a constraint solver engine.

Constraint programming is used in graphical systems to specify interdependencies between graphical elements. For example, in a diagram, a box may be connected by an arrow, and may contain text that is centered inside the box. Such positional relations may be specified as constraints on the positions between the box, arrow, and text. This has been used for both GUIs and VEs.

An example of a one-way constraint system is the Amulet GUI toolkit (Myers et al., 1997). Here, each GUI object is modelled as a node, which has a number of attributes with specific names, like VRML. Attributes may be added or deleted at runtime. Each attribute has a value, or a formula that determines its value. The formula is specified in terms of other objects' attributes, for example: $\mathit{DependentValue} = f(\mathit{value}_1, \ldots, \mathit{value}_n)$. Any changes in any of the values value_i implicitly generate events to update *DependentValue*. Further events will be generated as a result of the

change in *DependentValue*. Stated in dataflow terminology, the formula implicitly defines dataflow routes from the other objects' attributes to a script, which re-calculates the attribute each time some value update arrives.

Like VRML, Amulet provides a grouping construct, which may be used to define a system as a hierarchy of objects. Unlike the other models, Amulet's formulas may be explicitly specified in terms of the hierarchy. The following example (adapted from the Amulet manual) illustrates this:

```
Am_Define_Formula(int, owner_width)
    {  return self.Get_Owner().Get(Am_WIDTH);  }
Am_Define_Formula(int, owner_height)
    {  return self.Get_Owner().Get(Am_HEIGHT);  }
Am_Define_Formula (int, arc_width)
    {  return self.Get_Owner().Get(ARC_PART).Get(Am_WIDTH);  }
Am_Define_Formula (int, arc_height)
    {  return self.Get_Owner().Get(ARC_PART).Get(Am_HEIGHT);  }
Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 20)
    .Set (Am_WIDTH, 100)
    .Set (Am_HEIGHT, 100)
    .Add_Part(ARC_PART, Am_Arc.Create ("my_circle")
              .Set (Am_WIDTH, owner_width)
              .Set (Am_HEIGHT,owner_height))
    .Add_Part(RECT_PART, Am_Rectangle.Create ("my_rect")
              .Set (Am_WIDTH, arc_width)
              .Set (Am_HEIGHT, arc_height)
              .Set (Am_FILL_STYLE, Am_No_Style));
my_win.Add_Part(my_group);
```

The first four statements define some formulas returning integers. The first two return the width and height of the object's owner (=parent), the second two return the width and height of the ARC_PART attribute of the object's owner.

The rest of the statements define a group object with a circle and a rectangle inside it. The width and height of these are defined using the previously defined formulas: the circle gets its size from the group object, and the rectangle gets its size from the circle. If the group object's size were changed, the circle and rectangle would automatically adapt their sizes.

Amulet has another interesting feature that is worth noting. If, say, the rectangle were placed inside another group object, it would try to find an arc_part inside its new owner, and adapt its size to this new object's size. So, the formula is more than a dataflow route specification: it does not just define a fixed route between specific variables, but rather, it is a symbolic query that queries the node structure to find the desired values. As such, it is like a glue language. Such structure-based queries may be used to design components with some degree of flexibility: for example, Amulet defines standard interactor components that either operate on any of their sibling objects (analogous to

the VRML sensors), or, if they can't find any siblings, they operate on their owner instead.

Examples of a multi-way constraint system are Clock (Graham, 1995), which is a constraint language with GUI toolkit, and InViWo (Richard et al., 2001), which is a VE toolkit, which adds multi-way constraints to VRML worlds. Their manner of specification is similar to Amulet. Both have the concept of nodes, each having a number of attributes that can be read or written.

Conclusion. Constraints, like dataflows, are particularly suitable for programming complex graphics and animation. In the constraint programming languages we have seen, we also find a specific manner of structuring, namely, by identifying nodes which each have a number of attributes. Note that these are similar to entities in an ERD. While our proposed specification technique does not explicitly define value dependencies as constraints, the manner of specification is highly compatible with them, as our components may state interest in information by subscribing to it, and may then process updates at will. The approach is similar to the structure-based queries found in Amulet, though the kinds of queries supported are more general.

While multi-way constraints are interesting, their added value with respect to one-way constraints, their relative cognitive tractability, and their possible integration into the framework remain a matter of future research.

6.4 Control flow models

Control flow models are models that describe aspects of a system in terms of explicit control flow, that is, the specification of a process as a sequence of states. The transition of one state to another is specified as some kind of event or condition. More advanced models enable specification of multiple processes operating in parallel, which communicate in a certain manner. Most of them are easy to understand, executable, and usually automatically verifiable. However, they are limited in their expressiveness because they only describe a very small subset of the state space of a full system, requiring a large (and sometimes heuristic) abstraction step to be made with respect to the executable system. Still, finite-state specifications are usually well-defined, as they may carry all the way into a full system (Seo and Kim, 2001; Horrocks, 1998).

6.4.1 State automatons and statecharts

The best-known of these models is the finite-state automaton, and its slightly richer variant, the flowchart. State automata are widely used in software, and notably, they are found to be useful in NL dialogue systems, in which they are used to specify the dialogue structure (Bernsen et al., 1998).

A more complex control flow model, enabling hierarchical composition and some degree of parallellism, is the *statechart* model (Harel and Politi, 1998). Here, state ma-

chines may be embedded inside states of other machines, and entering a state means creation of the state machine inside it, and leaving means the destruction of it. While basic state automatons specify transitions simply as named events, statecharts specify them as a combination of event and/or truth condition. UML also adopts statecharts, and specifies that statecharts correspond to entities in UML structure diagrams, and transitions may be any statements about the state of the system, in particular the data structure as specified by the structure diagrams. This provides a well-defined coupling of statecharts to structure diagrams. Statecharts have also been used for VE development. The CLEVR methodology specifies VE objects in terms of statecharts, with the transitions specifying truth expressions in terms of system variables (Seo and Kim, 2001). In fact, the role that the statecharts play is similar to that of the Petri net specifications in Marigold.

Conclusion. Statecharts provide an interesting model in that they provide a clear coupling with a full executable system, including object creation and deletion dynamics. This does mean that they cannot be automatically verified, as their behaviour is determined by the behaviour of the system as a whole. Our proposed specification technique integrates statecharts, and shows how they may be used in a full executable system.

6.4.2 Process algebras

Process algebras are models for specifying parallel state automata. They provide mathematically simple models for composing multiple state automata into a larger system. Parallel state automata are specified in a way similar to statecharts. Unlike statecharts though, the composition of two state automata can be described by a new, bigger, state automaton. The reduction of any parallel system to a big state automaton makes it easier to prove properties of the system. Automatic verification tools are available for several variants of process algebras. Automatic verification is a particularly attractive feature of process algebras, because it provides a good method to find concurrency bugs in concurrent systems, and is one of the most important advantages of process algebras over less computationally tractable concurrency models.

The most popular process algebras are *Communicating Sequential Processes* (CSP) and *Calculus of Communicating Systems* (CCS). Both have been used in variations for the specification of GUI components and their interaction. Examples are usage of plain CSP (Alexander, 1990), and the ADC model which uses the CSP-based LOTOS (Markopoulos, 1997),

An informal description of the process algebras CSP and CCS will be given here. Process algebras describe systems as a number of parallel processes, communicating through channels which are identified by names. Note that the identification of processes and channels is similar in spirit to dataflow-oriented specifications. In fact, with respect to the part of the specifications that are similar to dataflow specifications, CSP and CSS may be seen as a limited type of configuration languages. A process may be created by another process, and may terminate itself. The dynamics of each process are described explicitly, in a form similar to that of a finite state automaton.

Communication between processes proceeds through channels, which are identified by names. To initiate participation in communication, each process signals that it is ready for a specific set of channels, called its nextset. The possible communications can be determined only when the nextsets of all processes in the system are known.

In CSP (Communicating Sequential Processes) (Hoare, 1985), communication is multi-way and symmetrical. In order to determine how many processes a channel is supposed to synchronise with, the set of channels that each process might participate in during its execution must be known when it is started. This set is called its alphabet, which may not change during the lifetime of the process. A process' nextset must always be a subset of its alphabet. Only when all processes that have a channel in their alphabet also have it in their nextset, communication over this channel may proceed.

In CCS (Calculus of Communicating Systems) (Milner, 1980), communication is two-way and asymmetrical. Each process is either a reader or a writer of a channel, and communication may proceed as soon as a channel has one reader and one writer. Therefore, it is not necessary for the processes' alphabets to be known in order to determine the system's behaviour.

In both CSP and CCS, communication is modelled as an atomic action, in which all participants participate simultaneously. All participating processes will then have to submit new nextsets. All non-participating processes will remain blocked. In case communication over several channels is possible, one channel is typically chosen randomly.

In CSP, data passing is not conceptually different from channel synchronisation. Data passed through a channel is modelled as an array of channels, obtained by concatenating the original channel's name to the representation of each value of the data's domain (for example, passing a byte could be represented by 256 channels, ending with the strings '000'...'255'). Writing a data value through a channel can then be modelled by requesting one specific channel out of the array, and reading though requesting any channel out of the array. A reader could specify only a subset of the values, thus forbidding any other values to be written. This last possibility may be seen as something 'in-between' reading and writing, since there is no fundamental difference between the two. The value or value range specified by each process may be instead seen as a *limiting constraint* on the value that is eventually chosen.

CCS has slightly different data modelling capabilities. Writers always write specific values. Readers may simply read any value offered, or they may specify a constraint on the offered value. If the constraint does not hold, the reader will refuse to read. Unlike CSP, another reader is still allowed to read the value instead. The value or value range specified by each reader may be seen as an *enabling constraint* on the value that is eventually written.

In both CSP and CCS, it is possible to create new processes recursively. In order to be really useful, this should be combined with some scheme to create new channels along with the processes. One scheme that is found in the CSP language are the channel hiding and renaming constructs. When a process is started, a set of channel names may be specified that may from then on be used for private communication between the started process and its creator. This is called channel hiding. For convenience, any

process may also have some of its channels renamed at startup. This scheme is limited, since the communication structure of the system may not be changed arbitrarily.

Example In this section, we give a large example of a CSP specification. This example describes aspects of the running example, and includes modelling of the different user interface objects, navigation through the environment, and representation of the dialogue structure of the dialogue agent. This specification has been simulated using some software written specifically for this purpose. It was also featured in a paper (van Schooten et al., 1999), which also proposed a means of coupling the CSP specification to an interactive prototype, and even combining it with a task model, also specified in CSP.

Parallel processes are specified using the parallelism `||` operator. `P1 || P2` means that P1 and P2 execute in parallel. Process behaviour is specified using the sequence `->` and choice `[]` operators. `a -> b` means that channel a is synchronised with, and then channel b. `a [] b` means that either a or b may be synchronised with. The alphabets of the processes are given implicitly by the channels they use.

```
//// The VMC VE consists of 5 main objects.
Vmc where
Vmc = User || Karin || InfoBoard || FrontDoor || InnerDoor,

User = UserObj,

//// Movement of the user.  There are three locations,
//// UserPos1...3. Proximity of the user to specific
//// objects is indicated by the userprox... channels.
UserObj = UserPos1,
UserPos1 =    (userproxkarin -> UserPos2)
           [] (userproxib    -> UserPos3),
UserPos2 =    (usernoproxkarin -> UserPos1),
UserPos3 =    (usernoproxib -> UserPos1),

//// The information board, which can be clicked on
//// (useribclick) to show more information.  The
//// information is extracted from a communication
//// with Karin.
InfoBoard =    IBObj,
IBObj = IBObjNoProx,
IBObjNoProx =    (userproxib -> IBObjProx),
IBObjProx   =    (useribclick -> initkarin2
                  -> textout3_perftime -> textin2_tellinfo
                  -> useribshowinfo -> exitkarin2
                  -> IBObjProx)
              [] (usernoproxib -> IBObjNoProx),

//// Karin is a very complex object, incorporating user
//// interface elements and a NL dialogue engine
//// (KarinEngine).
```

```
Karin =    KarinObj
        || KarinEngine
        || ChatLine
        || TextFrame
        || ResetButton
|| Table
//// the last four are just buffers for merging initkarin1..3,
//// exitkarin1..3, textout1..3, into resp. initkarin,
//// exitkarin, and textout, and splitting textin into
//// textin1 and 2.  These are not specified here.
        || MergeBuffer
        || MergeInit
        || MergeExit
        || SplitBuffer,

//// Karin shows its user interface elements as soon as a user
//// comes close, and hides them again if the user leaves.
KarinObj  =
      (userproxkarin -> openwindows -> initkarin1 -> KarinObj)
   [] (usernoproxkarin -> closewindows -> exitkarin1
       -> KarinObj),

//// The ResetButton, activated and deactivated by
//// userresetbopen/close,  reinitialises Karin when pressed.
ResetButton  = ResetBClosed,
ResetBClosed =  (openwindows -> userresetbopen -> ResetBOpen),
ResetBOpen   =    (closewindows -> userresetbclose
                   -> ResetBClosed)
                [] (userpushresetb -> exitkarin3 -> initkarin3
                   -> ResetBOpen),

//// The text frame shows the dialogue between the user and
//// Karin. The different types of utterances that Karin
//// distinguishes have been split into different channels,
//// similar to typical dialogue structure models.
TextFrame  = TextFClosed,
TextFClosed =    (openwindows -> usertextfopen -> TextFOpen),
              [] (closewindows -> TextFClosed),
TextFOpen =
     (closewindows -> usertextfclose -> TextFClosed)
  [] (openwindows -> TextFOpen)
  [] (textin1_hello -> usershowtext_hello -> TextFOpen)
  [] (textin1_tellinfo -> usershowtext_tellinfo -> TextFOpen)
  [] (textin1_telltable ->usershowtext_telltable ->TextFOpen)
  [] (textin1_askperf -> usershowtext_askperf -> TextFOpen)
  [] (textin1_asktime -> usershowtext_asktime -> TextFOpen)
  [] (textin1_confirm -> usershowtext_confirm -> TextFOpen)
  [] (textin1_done -> usershowtext_done -> TextFOpen)
  [] (textin1_cancelled ->usershowtext_cancelled ->TextFOpen)
  [] (textin1_error -> usershowtext_error -> TextFOpen),
```

```
//// The chatline enables the user to type text.  The
//// different kinds of requests understood by Karin are
//// distinguished as the different channels usertyped_...
//// The user may specify a genre (genre), a specific theatre
//// performance (perf), a date and time (time), or a
//// combination of both (perftime).  The user may book a
//// specific performance (book) and answer yes or no.
ChatLine  = ChatLClosed,
ChatLClosed =  (openwindows -> userchatlopen -> ChatLOpen),
ChatLOpen   =
     (closewindows -> userchatlclose -> ChatLClosed)
  [] (usertyped_genre -> textout1_genre -> ChatLOpen)
  [] (usertyped_perf -> textout1_perf -> ChatLOpen)
  [] (usertyped_time -> textout1_time -> ChatLOpen)
  [] (usertyped_perftime -> textout1_perftime -> ChatLOpen)
  [] (usertyped_book -> textout1_book -> ChatLOpen)
  [] (usertyped_yes -> textout1_yes -> ChatLOpen)
  [] (usertyped_no -> textout1_no -> ChatLOpen),

//// The table may show a list of theatre performances when
//// given the command usertablerefresh.
Table  = TableClosed,
TableClosed =    (opentable -> usertableopen -> TableOpen)
              [] (closetable -> TableClosed),
TableOpen =    (opentable -> usertablerefresh -> TableOpen)
            [] (closetable -> usertableclose -> TableClosed)
            [] (userclick -> textout2_perftime -> TableOpen),


KarinEngine =
//// These are variables, indicating the presence of certain
//// information in the dialogue context.
       ContextElem_genre
    || ContextElem_perf
    || ContextElem_time
//// This is the actual dialogue manager
    || MgrInit,

MgrInit      = (initkarin -> textin_hello -> MgrClear),
MgrClear     = (delperf -> deltime -> delgenre -> MgrWait),

MgrExit      = (exitkarin -> closetable -> MgrInit),

//// The manager waits for a dialogue move from the user, and
//// changes the dialogue context where appropriate.
MgrWait =
     (textout_genre -> addgenre -> delperf -> MgrAnswTable)
  [] (textout_perf -> addperf -> MgrPerf)
  [] (textout_time -> addtime -> MgrTime)
```

```
  [] (textout_perftime -> addperf -> addtime -> MgrAnswInfo)
  [] (textout_book -> closetable -> MgrBook)
  [] (textout_yes -> textin_error -> MgrWait)
  [] (textout_no -> textin_error -> MgrWait)
  [] MgrExit,
//// Determine the answer, depending on the dialogue context
MgrPerf      =    (gottime -> MgrAnswInfo)
               [] (nogottime -> MgrAnswTable),
MgrTime      =    (gotperf -> MgrAnswInfo)
               [] (nogotperf -> MgrAnswTable),
MgrAnswInfo  = (textin_tellinfo -> MgrWait),
MgrAnswTable = (textin_telltable -> opentable -> MgrWait),

//// Enter the booking sequence, which may result in a
//// successful booking (textin_done), a cancelled booking
//// (textin_cancelled) or an error (textin_error).
MgrBook     =    (gotperf -> MgrBookPerf)
              [] (nogotperf -> textin_askperf ->MgrBookWait),
MgrBookPerf =    (gottime -> textin_confirm -> MgrConfirm)
              [] (nogottime -> textin_asktime ->MgrBookWait),
MgrBookWait =
     (textout_perf -> addperf -> MgrBook)
  [] (textout_time -> addtime -> MgrBook)
  [] (textout_perftime -> addperf -> addtime -> MgrBook)
  [] (textout_genre -> textin_cancelled -> MgrWait)
  [] (textout_book -> textin_error -> MgrBookWait)
  [] (textout_yes -> textin_error -> MgrBookWait)
  [] (textout_no -> textin_cancelled -> MgrWait)
  [] MgrExit,
MgrConfirm  =    (textout_yes -> textin_done -> MgrClear)
              [] (textout_no  -> textin_cancelled -> MgrClear)
              [] MgrBookWait
```

Note that the dialect we used here was a minimal subset of CSP, that is, without channel hiding or shorthands for value passing. Basically, this means that a system is specified as a fixed number of parallel state automata. For the purposes of this general evaluation, this was considered sufficient. Two particularly interesting advantages we found were the ability to:

- Generate prototypes. A CSP description may be coupled to an implementation language, by making some CSP events trigger function calls, and translating some user input events to CSP events. This way, a prototype may be generated which illustrates both dynamical and appearance aspects.
- Do task modelling. CSP may be used to specify task hierarchies in a natural way, and is about as expressive as the most basic task modelling languages, such as GOMS. The task model may be executed with the system model.

However, we also found some limitations:

- Having to ensure that the processes have the correct alphabets with each incremental change during the specification process turned out to be a nuisance. When adding or temporarily commenting out particular processes, changes in other processes' alphabets sometimes needed to be made. A significant part of the errors made during specification were alphabet problems. This may not have been a problem in CCS.

  It may still be argued that the separate alphabet specification may be useful as a kind of double-checking, similar to type checking, enabling the computer to point out some programming errors clearly and effectively. However, unlike type checking, a process that specifies the wrong alphabet results in different system behaviour, rather than a clear deadlock situation or other kind of clear error indication. Furthermore, the only kind of double checking that may actually produce compiler errors is a mismatch between the process's alphabet specification (if given separately), and the actions used in the process's behaviour specification. Type checking is typically a cross-check between two specifications which are apart, such as a caller's call format and a callee's call format, reducing the need to keep browsing back and forth between the caller's and the callee's specifications. Alphabet checking only double-checks two specifications belonging to the same process, which is less useful.

- The multi-way synchronisation system was found to be cumbersome. The original reason why CSP was chosen instead of CCS is that CSP descriptions map more closely to logical constraints. This makes it easier to describe additional (limiting) external constraints on a given system by just adding extra processes, and makes it easier to transcribe the system to logical correctness specifications.

  However, it should be noted that the concept of multiple synchronisation constraints or any form of multi-way synchronisation at all was *not* used in the specifications that were actually written. This also means that the alphabet specification problems just mentioned were more of an inessential nuisance than a part of the fundamental problems of the systems being specified.

  Another problem was found while writing the CSP specifications: in several places, it was necessary to work around the CSP system in order to model 'one reader multiple writers' scenarios. These are very easy to do in CCS, as they fit naturally into its asymmetrical synchronisation model.

  The CCS model is closer to traditional procedural programming languages than CSP. This means that correspondence of the model with a final system may be easier to understand in CCS. In CSP, adding a process means limiting the behaviour of the rest of the system. In terms of procedural programming, this is very strange: when a process is *removed*, the rest of the program does *more* actions.

- The ability to model processes with arbitrary point-to-point communication channels is important, if not essential, for specifying systems with dynamical numbers of processes which have direct communication other than those between parents and children. Our specification does not include multi-user facilities. The standard constructs of channel renaming and hiding are not sufficient

to model this. Alternatives have been proposed to model this, such as the π-calculus (Milner, 1993). However, π-calculus is not model-checkable, so that it does not have the advantage of verification.

- The atomic communication model would have to be translated to a synchronous communication model in a full system implementation, which is inefficient in distributed environments. A first attempt to model asynchronous communication in process algebra using buffers indicated a state space explosion that precluded automatic verification. This means that the dynamics of a verifiable system may not be directly transcribed to the final system.

Conclusion. In retrospect, CCS might have been an improvement over CSP in terms of cognitive tractability. A language such as Promela (Holzmann, 1991) might be used, which provides convenient shorthands and a good model checker. However, the languages cannot be coupled well to a full system, as they remain insufficient to model distributed systems and systems with dynamical architecture.

6.4.3 Petri nets

Petri nets are a generalisation of state automatons, obtained by generalising the concept of state. In Petri nets, state is modelled by the existence of *tokens*, which move around the graph. A Petri net with a single token is similar to a state automaton. But, when tokens are passed through the net, they may be duplicated and consumed, so that any number of tokens may be in the system at any time. We may also have points in the net where tokens are created. Multiple tokens are a means to model parallelism, which, however, works somewhat differently from statecharts or process algebras, as synchronisation occurs through simultaneous consumption of multiple tokens. Unlike process algebras, there is no uniform way to compose two Petri nets into a larger concurrent Petri net. Instead, the nets will have to be connected using some heuristic. In some Petri nets, there may even be different kinds of tokens, or tokens may change state as they are passed through the system. Conceptually, a Petri net has characteristics of both a state automaton and a DFD. The tokens may be seen as data flowing through a DFD.

Petri nets may be used to model interactive systems. They may be combined with an executable system, or may be automatically verified. For example, the article (Palanque and Bastide, 1996) describes how Petri nets may be used to model both human and computer activity in a human-computer system. Petri nets are used in several interactive systems specification techniques. One system we have already mentioned is the Marigold system, which combines them with dataflows to obtain VE object behaviour specifications. Another system is the *Interactive Cooperative Objects* (ICO) specification technique (Navarre et al., 2001; Palanque and Bastide, 1995b; Palanque and Bastide, 1995a), which uses Petri net to model the behaviour of objects, which is much like Marigold. The Petri net specifications are integrated into a full system using a software tool called Petshop.

Conclusion. In the systems we mentioned, Petri nets play a role in VEs or GUIs that is similar to state automata, namely, that of the description of objects. While Petri nets are interesting, they provide no clear advantage over statecharts or process algebras. Rather, their lack of compositional structure is a disadvantage. The combination of Petri nets with dataflows as proposed in Marigold is interesting, but there may be good alternatives, such as combining state automata with dataflows, or using constraints in the place of dataflows.

6.5 Conclusion.

We have given a short review of a number of interesting languages that have been used and may be otherwise suitable for specification of the different aspects of VEs. While most languages obviously have their merit, we have indicated the advantages and disadvantages of each with respect to our own requirements, in which we put some extra emphasis on well-definedness and computational tractability. We summarise our findings in the table below.

In the next chapters, we describe a specification technique that is effectively based on a selection out of these classes of languages. Central is system architecture, for which we use a glue language approach, combined with structure diagrams. In addition we use statecharts to be able to easily specify some basic dynamical behaviour. We chose not to use dataflows, process algebras, and Petri nets, which may have been chosen as alternatives to describe these aspects. Dataflows and process algebras are lacking in their ability to describe dynamically-changing structures, and Petri nets are lacking in describing compositional structure in general. Hence, these three language types are lacking in expressiveness and well-definedness.

To specify further details of (components of) the system under development, such as detailed system properties and complex behaviour, we deemed predicate logic, logic programming, and constraint programming to be most interesting. Of these, we applied predicate logic for system correctness constraints, and listed the latter two as future development. The alternatives, temporal and intentional logic, were deemed less interesting. Temporal logic has limited expressiveness and well-definedness, and intentional logic has limited computational and cognitive tractability.

Name	Expressiveness	Well-definedness	Computational tract.	Cognitive tract.	Usage
dataflows	low	full system	simulation	nonexpert	architecture of GUI, 3D
glue languages	varies	full system	simulation	varies	architecture of GUI, 3D, multi-user
structure diagrams	low	full system	assertion checking	nonexpert	domain data and architecture
constraints	high	full system	simulation	expert	GUI, 3D
state automata	low	full system	simulation or model checking	nonexpert	NL dialogue, single VE object
process algebras	low	simplified system	model checking	expert	architecture including NL dialogue and VE objects
Petri nets	low	full system	simulation or model checking	non-expert?	VE objects
temporal logic	high	simplified system	simulation	expert	various
intentional logic	very high	ill-defined	type checking	expert, prohibitive	NL dialogue, agent behaviour
logic programming	high	full system	execution	expert	NL dialogue, agent behaviour
predicate logic	high	varies	type checking or assertion checking	expert	any aspect

Expressiveness is roughly indicated, with *low* meaning only a rather limited aspect of the subject can be expressed, and with *high* a relatively full coverage is possible.

Well-definedness is specified w.r.t. the executable system. Most languages directly correspond to the full system, but there are a few notable exceptions to this.

Computational tractability is classified into the general classes we defined in chapter 5: syntax checking, assertion checking, theorem proving, simulation, and model checking.

Cognitive tractability is difficult to determine exactly. We mostly limit ourselves to a classification into *nonexpert*, meaning it is usable by a non software expert, and *expert*, meaning it is likely to be usable only by a software expert. This classification is essential for determining what class of developers the specification languages are suitable for.

Part III

The VETk technique

Chapter 7

Fundamentals of the VETk technique

7.1 Overview

In this chapter, we will introduce our proposed specification technique. We will give a rationale and an overview of the languages in the technique that should be sufficient to understand the examples in chapter 8. A more detailed and concise overview of the technical details of the language is found in appendix A. We assume that the reader has knowledge of the HTML, Java (Zukowski, 2002), and Python (Beazley, 1999) languages.

In chapter 6 we have made a first assessment and selection of interesting languages, with each language being especially useful for specific aspects of VEs. Many of the different languages and toolkits we have seen are useful for our purpose, but they need to be integrated to cover all identified aspects of VEs (graphics, multimodality, interface agents, and multi-user). This is why we will start with our own technique: we wish to find out how these aspects may be integrated. The specification technique plus software toolkit that we will introduce here will be called *Virtual Environment Toolkit (VETk)*.

We chose to create such a full-blown specification technique for several reasons. One is that we have identified the design of a specification technique as a difficult problem, and we require examples, experience, and a appropriate way of thinking to properly use, and hence, assess a new specification technique. Another is that we have stated executability as one of the most important criteria, and that only a fully implemented executable system will show whether our ideas really work.

We chose to develop full languages with their own syntax, rather than casting the specification technique in the form of software libraries. This is because conciseness is significantly increased this way. In fact, part of the specification technique started off as software libraries, but a need for conciseness led to the development of the speci-

fication languages. The languages are derivatives of existing languages. While some may consider learning new language syntax rather than using software libraries in a familiar language an extra psychological disadvantage, we believe that the disadvantages by no means outweigh the advantages:the amount of new syntax to learn is limited and only has to be learnt once, and the most difficult part of the specification technique, the semantics, is similar in complexity in both cases.

In the current version of our framework, we will emphasise software architecture. We will choose a component-based architecture, with GUI objects, VE objects, and interface agents being the components. We wish to enable integration of the best of existing ideas. We argue for the use of multiple languages. The languages need to be integrated, and the model thus formed should preferably be extensible to incorporate more languages. We will pay special attention to how the languages fit together. We use a glue language approach (see section 6.3.2), and integrate executable code with ERDs, state automata, and logic assertions.

We will include a bit of development history here. Our main interest at first were process algebras, and we built an execution engine with prototyping facilities based on them (van Schooten et al., 1999). The need for dynamical creation and modelling a fuller version of the system became apparent however, and a new technique was conceived. After a first version, which implemented the rudiments of the database idea (van Schooten, 2000a; van Schooten, 2000b), was soon discarded, we smoothed the database idea by using a proper database engine and cleaner, more powerful syntax (van Schooten, 2001; van Schooten, 2002).

We chose a Web environment as our execution platform, using Java and HTML as our basic platform. While this imposes some technical idiosyncrasies that are not relevant to our research, the successful implementation of the technique in this environment does provide an interesting proof of concept of full system modelling using the technique. Note that our in-house Virtual Music Centre project also runs in a Web environment, which is one of its project goals, and we wish to have a system that is usable in a similar environment. Besides, we consider Java a convenient programming language and basic platform, and the step towards a Web environment was not such a large one, especially since HTML proved to be a useful language as well.

7.1.1 Modelling structure using ERDs and a shared database

The basis of our technique is the use of explicit data structure specifications. Like ERMIA, we advocate the use of structure diagrams (namely, ERDs) for the structure of the user interface. This has not been done often, and we have argued that this is an interesting direction. We integrate this approach with executable specifications, so that the specifications are well-defined. Entities correspond to software components: each software component is assigned a unique entity ID. Furthermore, we use a database approach, including a database view facility, to access the data structure. This has the advantage of easy access of the required data in the case of complex information requirements. The database is similar to a traditional ‘relational’ database, enabling SQL-type queries, but is special-purpose with respect to user interface data, in the

sense that it has a very low latency, but is not efficient for bulk data. The view facility consists of the ability to specify a query, of which the results are updated automatically when a change is made to the database. This is a special case of a publish-subscribe facility.

Beside specifying user interface structure, we may specify user interface *state* in ERDs by means of entity attributes. Note that we have already discussed some user interface toolkits that use such an approach, using the concept of nodes with attributes in them: they are Amulet, Clock, and VRML. They are based on tree structures rather than E/R structures. The component structure is a tree, and each component is able to define variables within its own node within the tree. Component interaction may be achieved by traversing the tree structure and reading nodes' variables as needed. Of these three, Amulet comes the closest, as it enables its formulas to be specified by means of queries specified in terms of this tree structure.

To summarise, our approach has the following potential advantages:

- **Usefulness for usability aspects.** Structure diagrams are readable by non software engineers, which makes them potentially useful for evaluating usability aspects. As we have seen in the Netscape find box example in section 6.3, an ERD combined with screen layouts may be used to evaluate visibility issues.
- **Specifying component structure in an abstract yet comprehensive way.** Structural specifications in terms of E/R models may bridge the gap between first design stages and a full implementation of a component structure, by making clear the pattern that the structure should conform to.

  This is especially suitable for modelling the dynamical creation and deletion of components. We gave an example of a multi-user editor component structure as a typical 'dynamical' component structure that we should model using structure diagrams (in our case ERDs). Furthermore, using the database view facility, the components are naturally able to deal with dynamically-changing sets of values as input.
- **Specifying navigational structure.** Several techniques exist that specify the structure of a hypertext system as a data structure. A similar technique may be adopted for VEs. The structure diagram can specify the structure of a virtual world.
- **Enabling visibility for interface agents and multi-user applications.** We may assume that especially interface agents and multi-user applications may benefit from this approach, as in either case, visibility and awareness of the various elements of the user interface data structure are important. While ERDs are useful to determine visibility issues in many interactive applications, they may be especially useful for these classes of applications. The database view approach should make it easy to specify which information should be extracted from the data structure.

  In fact, this structuring concept forms an interesting basis for 'grounding' interface agents in their environment (Robert et al., 1998) using a structured symbolic

representation, which forms a suitable abstraction from the physical (i.e. graphical) level that humans use. A similar use of Prolog-like facts to describe a world structure is also found in (Anastassakis et al., 2001).

7.1.2 Distributed usage

As a multi-user technique, the system will need to be suited for distributed usage, which is a natural responsibility of the glue level. Our basic approach is that the shared database works like a distributed shared memory. It is accessed in a uniform way by all components, and furthermore, it is the only way through which agents can communicate. This way, the technical step of going from a single-user to a multi-user system is very small.

Ideally, a system needs to be both easy to program and scalable to a large number of computers working over unreliable long-distance networks. Here, we try not to solve all of these distributed systems issues, as these are notorious. While it is known that such distributed systems issues may have an impact on user interface design (Bhola et al., 1998), this does not mean that a system that does not solve all of the problems is not useful. Since our emphasis is on architectural modelling here, the distributed systems problems are left as much as possible as an 'implementation detail' that may in part be solved by a better implementation of the same execution model. Still, we do want some degree of distribution to be possible, and use a limited subset of techniques to achieve that. Our basic approach is very simple: we use asynchronous communication with a maximally restrictive message ordering scheme, which is basically: all recipient receive messages in the order in which they were sent. This provides a simple and workable trade-off, suitable for local area networks. In terms of distributed systems, we provide a publish-subscribe scheme (in the form of automatically-maintained database views), which is inherently asynchronous (Eugster et al., 2000), and an asynchronous remote procedure call/message passing scheme (in the form of events, which are modelled as a special kind of attribute that is suitable for passing a message).

A fuller implementation could for example be made by enabling different schemes, such as synchronous and asynchronous, and the addition of real-time facilities for 'streaming' media. This way, the implementors can choose their own scheme for each particular case. For now, we use one uniform scheme, which should at least be acceptable for proof-of-concept or prototyping purposes.

The loose coupling between agents by means of queries, as well as specifying neat agent exit behaviour, facilitates robustness against agent crashes and network disconnections. For example, it is not particularly difficult to model users as being able to exit or disconnect at any moment in time. In the spirit of open systems, in many cases, the system may keep running while some particular parts of it is under development.

7.2 Combining languages into a technique

In this section, we will introduce the different languages and how they fit together. We have three classes of languages: system properties constraints (specifying properties that the desired application should conform to), glue languages (specifying how a set of components is glued to specify an application), and component behaviour languages. Below is a table of the kinds of specifications currently supported by the framework. We distinguish three different kinds of specifications, as given in bold in the table. At the left side are the kinds of specifications that are supported, at the right side are the corresponding languages.

System property constraints	
ERD of user interface	VDC (VETk Data Constraints)
State automata of UI objects	VDC
logic constraints on any data in database	VDC
Component glue	
Main glue language	VETkScript
Glue language + visual layout	HTML + VETkScript
Individual component behaviour	
Component interface	VCL (VETk Component Language)
Component behaviour	Java

These languages will be introduced in the next sections.

7.3 Execution model

Before we start with the specification languages, we will discuss the overall execution model of the system. The system is based on an engine that implements the database and scripting facilities. It runs as a separate process. Agents interact with the engine after establishing a TCP/IP connection. We enable agents to be run both as stand-alone Java applications and as applets in a Web page, with each agent being a separate applet. Both applications and applets may open windows using the appropriate VETkScript commands, while applets may additionally load HTML pages (with further agents embedded in them) through a request to the browser. A separate Web server serves the relevant HTML and VETkScript files. Java requires that TCP/IP connections by applets can only be made to the same machine as the Web server, which means that the engine is required to run on the same machine as the Web server. See figure 7.1.

The engine receives requests to register new agents and unregister deleted agents, and executes VETkScript scripts. A client machine (a Java Virtual Machine) has exactly one TCP connection with the engine through which all messages are sent and received. This means that all messages between the engine and the client machine arrive in the order they were sent. The client machine runs a number of agents, and has a queue storing incoming database query results from the engine. Each agent has one Java

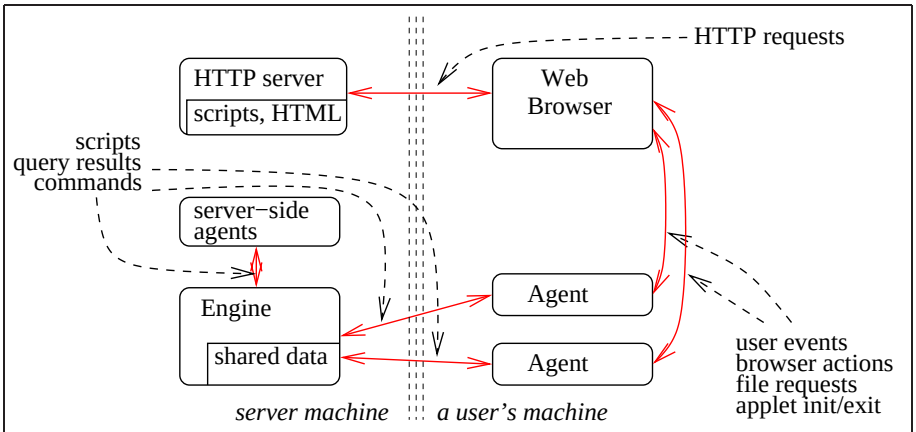


Figure 7.1: Architecture of system in Web environment.
The figure shows a user logged in using a Web browser. Beside running the Web server and the engine, the server machine also runs some agents here, which may for example be domain agents or interface agents. The lines indicate communication links.

thread executing the VCL code. The VCL execution consists of processing the results from the queue one by one, and stops when the queue is empty. Other threads may run on the agent's machine (such as a GUI event handler thread or timer threads), which may communicate with the agent through internal messages, which are added to the agent's queue and handled by a separate VCL message handler. See figure 7.2.

The engine executes scripts asynchronously: when a script sent by an agent arrives at the engine, it is queued before actually being executed. The engine executes scripts one script at a time. During the execution of a script, one or more messages may be generated, which are immediately sent through the TCP connections to the appropriate client machines. They will arrive in the order in which they were generated, though the length of the delay is undefined, and may be different for different client machines. The commands that generate messages are the following: registering a new query to the agent (@addquery), creating a new agent (@new), and commands for managing windows on the agent's machine (@window, @wadd, @wclose) or loading a HTML page (@execute).

7.3.1 Agent initialisation, exit, and update notification

There are two ways to start an agent: by means of a VETkScript @new command, and by loading a Web page containing the agent. When an agent is initialised, it is assigned an entity ID, its initialisation script init_s is executed, before it starts handling messages. In case the agent is loaded from a Web page, it also knows the IDs of the other agents in the Web page, making it easy to address them.

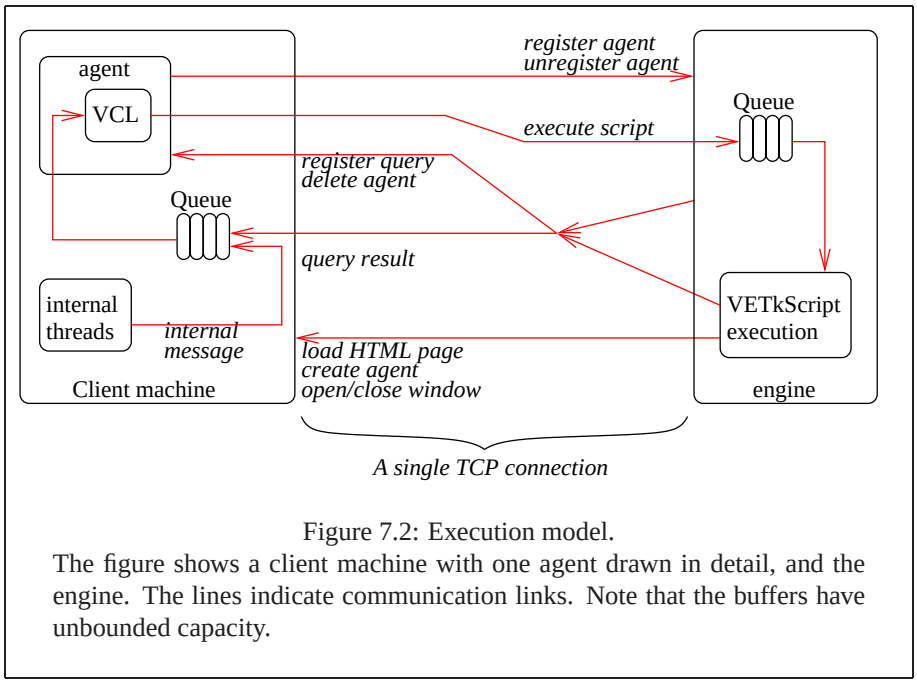


Figure 7.2: Execution model.
The figure shows a client machine with one agent drawn in detail, and the engine. The lines indicate communication links. Note that the buffers have unbounded capacity.

When an agent exits, or is deleted or disconnected, all the database information related to its ID is deleted as an atomic action.

There are three kinds of database update commands: additions, deletions, and events. Additions and deletions are issued immediately, as atomic actions, and the difference of the database before and after the action is used to determine the update messages to send to all subscribed agents. Events, which may be used to model regular message passing, are modelled as an addition immediately followed by a deletion.

A single update may consist of several changes to several views. An agent may react to an update in several ways: each change in a view triggers a corresponding function call in the agent, with all deletions being handled before all additions. Furthermore, special functions are called at the beginning and the end of the update that may contain extra code to handle the change as a whole.

7.4 The glue languages: VETkScript and HTML

7.4.1 Glue language approach

In our component-based architecture, we use a glue language approach, based on database queries and operations, to integrate components into an application. Components (which are called *agents*) are defined in a way that is similar to dataflows: each has a number of input and output points. Data arriving at an input may trigger agent

behaviour and arbitrary output. Components interact through a shared database. Their inputs are coupled to the database by defining database views. Their outputs are translated to database operations by means of the execution of scripts in the VETkScript language (see figure 7.3). These are defined when a component is created. In order to correspond to views, the inputs of the components are *sets* of values, with changes to these sets being notified. Such a fundamental support for sets as input is useful for a variety of things. Examples are user interface components that deal with sets of elements, such as listboxes (producing a clickable list of values), canvases (displaying a set of objects), and multi-user components, such as multi-user scrollbars (displaying a scrollbar for each user). In fact, the idea of coupling database views directly to user interface views is well used in Web interfaces (Moulding, 2001) and it has even been proposed to use OCL to specify user interface views. In these cases however, the database contains only (relatively static) domain data, rather than user interface data (i.e. the structure and state of the user interface).

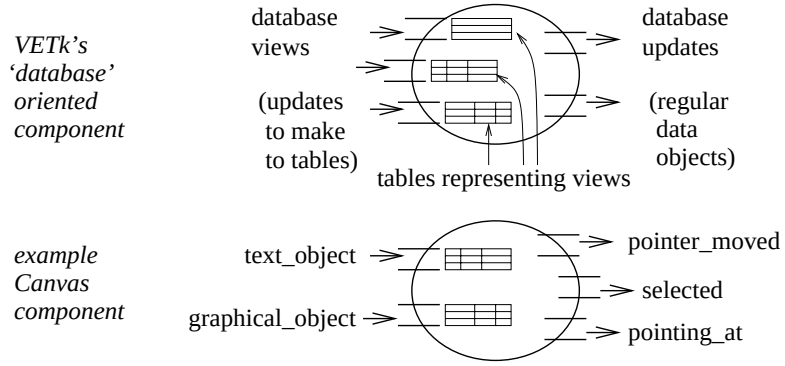


Figure 7.3: The VETk agent.
The top figure illustrates the 'database' oriented nature of a VETk agent. An agent has a number of views (which correspond to database tables, or lists of tuples) and output points (with each output point corresponding to a tuple with specific size and element types). The bottom figure shows an example agent, a GUI canvas, that displays the sets of objects given by its views *text_object* and *graphical_object*, and produces output generated by user interaction: *pointer_moved*, which outputs new coordinates of the mouse pointer, and *selected* and *pointing_at*, which provides information about objects that are pointed at or clicked on by the user.

When viewed from a cognitive viewpoint, such a glue specification has the potential of being properly concise and self-contained. The explicit separation into two levels should encourage the separation of specifications into highly application-specific code (i.e. the glue), and potentially generic and re-usable code (i.e. the agents). Once the agents are known by heart, the glue specification becomes self-contained apart from the non-standard agents. The specialised nature of the glue language should enable the

glue specifications to be appropriately concise.

Visual layout with programming languages

We enable the mix of visual layout specifications with glue, enabling both visual structure and the interaction of these visual components to be specified in a single specification. There is an increasing number of specification techniques that combine visual layout specifications with programming languages. Visual layout specifications typically take the form of layout constructs similar to textual layout constructs (such as tables, line breaks, sections, centering, etc.) or general visual relations, such as embedding. In some languages, syntactical embedding corresponds to visual embedding or visual composition (i.e. how a whole is composed into parts).

Examples of such languages are VRML (Carey and Bell, 1997) and HTML. In particular, HTML has an interesting history in this respect. The original HTML only allowed embedding of behaviour by means of Java applets, which are allotted their own window within the HTML page to display their output to. However, Java allows little interaction with the HTML page itself, and the need to further control the appearance and behaviour of the Web pages lead to the ad hoc introduction of Javascript, which allowed the HTML page to animate and react to user actions in an arbitrary manner. Javascript is a rather traditional programming language that is embedded in special fields inside the HTML code. Javascript has access to the Web page structure as a data structure. In fact, the HTML code may be seen as data structure definitions that can subsequently be manipulated by the Javascript.

After Javascript, a second kind of language made its ad hoc entry into Web programming: languages to generate HTML code by adding programming facilities into HTML pages. For example, a loop surrounding a piece of HTML code may be used to repeat that particular piece of HTML multiple times. This particular scheme ensures that the ‘degenerate’ case of a HTML generation program as a plain HTML page is possible, so that a natural mix of plain HTML with (partially) generated HTML is possible. Such a layout generation language could also be useful for other layout languages, such as VRML. This ‘mix’ approach may be seen as a trade-off between concise and comprehensive layout notation and appropriate constructs for generating the layout. Examples of these languages are the well-known PHP, mainly used for generating HTML, and QTk (Grolaux et al., 2001), which combines program code with layout statements, meant for generating GUI layout. While we do not enable layout generation in our current technique, this is an interesting topic of future research.

The debate on whether visual layout is better described using a direct visual representation has not been concluded. This issue is apparent in word processors, which use either a visual representation that closely matches the final output of the document being edited (called What You See Is What You Get, though some of its opponents call it What You See Is All You Get) or encodes the layout of the document using directives that determine constraints that the desired layout should conform to. The addition of programming language constructs may tip the balance towards the latter, though means to improve the comprehensiveness of the various existing languages, such as HTML,

Javascript, PHP, or VRML using a different (visual or non-visual) syntax is a topic of future research. Currently, we use a textual specification, but it may also be possible to use a visual representation to specify both. For example, (Banavar et al., 1998) describes a visual language that combines visual layout with dataflows.

7.4.2 The languages

In this section, we will introduce the details of the VETkScript language and its embedding in HTML. VETkScript specifications will be called *scripts*.

Scripts are called at specific points in each agent's control flow. An agent is integrated in its environment by defining the content of these scripts, which must be written in VETkScript. In each agent, one may define a startup script *init_s*. This script may be used to configure the agent's data, define its database views, or launch other agents.

VETkScript is basically Python with some database extensions. Python is an object-oriented language, which allows code to be compiled and executed on the fly, and thus lends itself well to 'scripting' approaches. Python also has special facilities for working with sets and lists, which makes it well suited for database operations. We use the Jython implementation here, which is Python written in Java. Jython integrates well with Java. The Jython engine has been modified slightly to cache the results of code compilation, so that repeated execution of identical source code does not require re-compilation, obtaining, in our case, a speed improvement of about a factor of 50.

A script has access to a set of variables. These variables are part of the agent, and remain defined as long as the agent executes. When an agent starts, it always sets some special variables which have a fixed meaning. New variables may also be defined when the agent executes a script using the VCL execscript. There are several other situations in which variables are defined, which are described below.

HTML may also be used to define a visual layout containing (GUI) agents by embedding the agents as applets, using applet parameters to configure the agents. The agent IDs of any named applets in the Web pages are available as script variables to all other agents in the page.

While HTML with applets is not an ideal language for defining agents in a visual layout, it was easy to implement as it makes optimal use of existing facilities. The conciseness and ease of use of HTML could be improved, in particular, defining applets in a HTML page is somewhat verbose. Still, HTML specifications has been used successfully, and was found acceptable for our purpose as well. One problem with applets is that they cannot define their own window size, and the size has to be set in the HTML code, which is less ideal. Even though there are functions to set the applets' size, they are typically not implemented in Web browsers. Another problem we ran into was that the behaviour of applets in different browsers is different, and that some browsers contain serious bugs in their applet handling. We developed our applications to work with Netscape and IE.

7.4.3 Storing objects in the database

Before we continue with the rest of the VETkScript specification, we will explain the structure of the database and the syntax used to describe tuples in it before continuing with the VETkScript specification. The database is a Prolog-style database that stores information as predicates (also called tuples). They have the following form:

```
<tuple_name> '(' <parameter> ',' <parameter> ','  ... ')'
```

Each tuple has a name and one or more parameters. The parameters are weakly typed, i.e. no type needs to be assigned beforehand. Each tuple with a different name, one different value in a specific parameter position, or a different number of parameters than another is considered a different tuple. Adding the same tuple a second time means nothing happens.

The database distinguishes two types of objects that it can store as predicate arguments: primitive types and object references. Values of primitive types are those that can be pattern-matched by the Prolog engine. These are currently limited to Strings and Integers. Other Java types are stored as object references, which may not be matched with literal values or other parameters.

A special exception is the Agent (agent ID) class. An argument of the Agent class is stored as an Integer. Additionally however, the database keeps track of all tuples that contain Agent arguments, so that when an agent exits, all tuples with its ID in them will be deleted automatically. The agent's entity, and all its attributes and relations are thus deleted. This was found to be useful default exit behaviour. A future version may incorporate customised exit behaviour, for example, an *exit_s* script might be defined that is executed by the engine when the agent is removed.

7.4.4 Transfer of data over the network

Network communication mainly occurs when scripts are executed and when database result updates are passed to each agent. A caching scheme is provided that improves efficiency in some cases (it is in fact inherited from the underlying ObjectStream library). In this scheme, objects are cached by reference ID, so that, when the same object is passed twice, only a reference needs to be passed the second time. Unfortunately, this scheme also works against you in some cases. When an object is passed, then modified, then passed again, the other end receives the old value of the object. In our implementation, the cache is flushed periodically to prevent memory leak, so that passing the same object twice does not guarantee that the same reference is received at the other end.

In order to avoid semantical problems, all objects passed as script variables should be value objects. A value object is an anonymous object of which only the content matters (i.e. its reference may not be compared to other references), and which may not change its contents during its lifetime (i.e. an immutable object). Integers and Strings are examples of value objects.

7.4.5 Language structure

Basically, VETkScript is just Jython with some extra commands. The only real change made to the Jython language is the replacement of the scoping marks. Jython uses indentation to indicate scoping, that is, an increase in indentation level indicates a new scope, while a decrease in indentation level, back to the previous level, indicates the end of the scope. VETkScript uses the symbols '{{' and '}}' to indicate the start and end of a scope. These symbols should be used in exactly the same places where indentation changes are normally used, resulting in a scoping scheme similar to C or Java. This change was made for two reasons. The first is that indentation is a dubious way to indicate scoping, as one is forced to indent everything as the rule prescribes, and the usual alternative, using braces, is only slightly more verbose. The second reason is that VETkScript is also passed as applet parameters, and applet parameter passing typically destroys whitespace information. A second minor change is the addition of Java-like comments, '//' and '/* */'

For each script that is executed, several variables are defined. Variables remain defined during the lifetime of the agent, unless overwritten. One is the agent's entity ID, called *self*, its window ID *self_window*, and the IDs of other agents on the same HTML page, in case the agent was started from a HTML page. Other variables are the parameters that the agent passes to the script when it calls the script: this is the main method for passing information to the script to enable the script to process it appropriately. When a new agent is started, it inherits its variables from the other agent, so that its 'parent' may pass information to it when necessary. A copy is made of the complete variable space, so that agents never share any references in scripts, and hence, cannot access shared data through script variables.

VETkScript defines a number of actions and functions. We give a short account of them here, with more syntactical details featured in appendix A. Actions have a different syntax depending on the kind of action. Functions take bracketed parameters, like regular functions.

Actions comprise the database actions and the creation of new agents. We find predicates clauses in the addition and deletion actions (indicating tuples to add and delete), and in queries (indicating a Prolog-style search pattern, as we have seen in section 6.2.3). Each clause is a list of database tuples, as defined in section 7.4.3. In queries, we may use variables as tuple parameters, indicating the variables to be bound to the values found in those parameters. Query variables are indicated by prefixing them with a tilde ~ symbol.

Ex. the query expression *~y : tuple1(~x, 1+1, ~y) tuple2(~x, "foobar", _)* indicates all values at the third position of the 3-tuples 'tuple1', as indicated by the query variable ~y. These tuples should have a '2' at the second position. The value at the first position should match the value at the first position of at least one other 3-tuple called 'tuple2', as indicated by the query variable ~x occurring at these positions, indicating a pattern match similar to an SQL *join* operation. Finally, the value at the second position of the tuple2 tuple should be the literal string 'foobar'. Say, if the database contains the tuples *tuple1(555, 2, "ape"), tuple1(555, 2, "banana"), tuple2(555, "foobar", "don't care"),*

the query result is the set of 1-tuples *("ape"), ("banana")*.

Note that we are still missing a 'not' operator in our queries, to indicate the absence of a tuple as a requirement for a query result. The precise form of such an operator is a matter of future research. The language is also lacking a '@delete' command to delete an agent. For now, we can use @wclose (close an agent's window) to delete an agent.

With the *@new* command, we may create new agents or new entities:

- **@new** <**id_var**> <**agenttype**> <**agentparameters**> - create new agent of type agenttype, store its ID in *id_var*, and store its window ID in *id_var_window*.
- **@new** <**id_var**> - create new entity key and store its ID in *id_var*.

We can define subscriptions by means of the *@addquery* command. This couples a query to an agent's view. The query's results, as well as any updates made to it later, are sent to the agent, so that it can update its internal view table accordingly. Multiple queries may be coupled to the same view, resulting in a union of the queries' results.

- **@addquery** <**view_name**> <**parameterlist**> **:** <**predicates**> - Couple a database query to the view of this agent called *view_name*.

  The *parameterlist* is a comma-separated list of Jython expressions, in which any query variables bound in the predicates may be used.

  From each element in the query result of the query defined by *predicates*, the query variables occurring in the *parameterlist* are extracted and substituted by the result, resulting in a set of tuples that represents the view.

  Ex. the action *@addquery position_view ~AGT,~X,~Y : position(~AGT,~X,~Y)* couples a query to the view *position_view*, which represents a set of 3-tuples specifying agent ID, X position and Y position.

Additionally, there is a command for querying the database directly so that the query results can be processed in the script:

- **@doquery** <**parameterlist**> **:** <**predicates**> - Do immediate query and return the results as a list of tuples.

  This direct query action enables a script to extract any necessary information from the database.

Then there are the actions to do updates.

- **@update** <**action**> <**predicates**> - Add or delete predicates. action may be one of '+', (addition) '-', (deletion) or '.' (event, an addition followed by a deletion).

  This is the basic database update operation.

- **@updateq** <**action**> <**predicates_to_upd**> **:** <**query_predicates**> - Add or delete pattern. *predicates_to_upd* may contain variables defined in the query *query_predicates*. For each result of the query, the variables corresponding to the result are filled in in *predicates_to_upd*.

  This operation enables updates to existing information to be done easily, or generally specify updates according to information in the database.

  Ex. the expression *@updateq + position(self,*~X,*newy) : position(self, ~X, _)* specifies that an agent's position attribute is updated. The X position is taken from the original attribute (*~X*), and the y position is set according to the script variable *newy*. Note the use of the *non-key prefix* '*', which indicates that the new position value should replace the old position value.

Beside the actions, there are some functions which take regular parameters as regular functions do. The most important are **@execute** that executes a VETkScript or HTML script, and window managing functions **@window**, **@wadd**, and **wclose**.

7.4.6 Example

This example is a minimal chat application, which consists of a single HTML specification, embedding standard GUI agents. Users can enter by loading the HTML page. When they do, a dialog box appears in which they can type their name, and then they can commence chatting by typing text in the text field at the bottom of the page. At the top is a text area which shows everything that's going on in the chat room: users entering, leaving, and chatting. The data structure is very simple: there is only one entity *minimalchat_user*, which has an optional attribute *user_name*, and one event *speak_event*.

The entity *minimalchat_user* is modelled as a tuple with as its first parameter the entity's key. Note that the optional attribute and the event is modelled by means of separate tuples which have as their first parameters again the key of the entity they belong to. This manner of modelling is used throughout the specifications.

```
1  <HTML><HEAD> <TITLE>Minimal Chat application</TITLE> </HEAD><BODY>
2  <!– The area where all information is displayed –>
3  <applet code="TextAreaAgent.class" width=600 height=400 align="middle"
4  name="user_agent">
5  <param name="columns" value="90"> <param name="rows" value="30">
6  <param name="init_s" value='
7       /* Log the messages of users.  This is done by the append view, which
8        * reacts to additions by appending any of their content to the text
9        * displayed in the text area.  So, the purpose of the view is to listen
10       * to events rather than maintain a set: it is actually a message
11       * handler. */
12      @addquery append ~NAME, ": ", ~TEXT :
13           minimalchat_user(~USER)
14           user_name(~USER, ~NAME) speak_event(~USER, ~TEXT);
```

```
15      /* Show entry of new users.  Each element that is added to the specified
16       * set (i.e. a user is created) results in the display of a message. */
17      @addquery append "User ",~NAME," entered." :
18          minimalchat_user(~USER) user_name(~USER,~NAME);
19      /* Show exit of users.  This is the reverse of the last query: a message
20       * is displayed for each element that is deleted from the set. */
21      @addquery append_deletion "User leaves." :
22          minimalchat_user(~USER);
23  '> </applet>
24  <BR>
25  <!– The field where text can be entered –>
26  <applet code="TextFieldAgent.class" width=600 height=35 align="middle">
27  <param name="columns" value="90">
28  <param name="init_s" value='
29      /* announce the arrival of a new user */
30      @update + minimalchat_user(user_agent);
31      /* create a window where name can be typed */
32      @window("win",130,80,1,2);
33      @new nlabel LabelAgent width=100,height=35,text="Your name:";
34      @new ntextf TextFieldAgent width=100,height=35,
35          text_out_s = <<
36              /* define user_name when user enters text */
37              @update + user_name(user_agent, *text);
38              /* now, the window closes itself and the agents are deleted */
39              @wclose(self_window);
40          >>;
41      @wadd(win,nlabel_window); @wadd(win,ntextf_window); @wadd(root,win);
42  '> <param name="text_out_s" value='
43      /* when the user types text, communicate this as a speak_event */
44      @update . speak_event(user_agent, text);
45  '> </applet>
46  </BODY></HTML>
```

When the page is loaded, the two *init_s* scripts are executed, initialising the agents' views, and creating a window in which the user can type his/her name (lines 32-41). When the name is entered, the user can chat by entering text in the TextFieldAgent. When text is entered, the Java GUI library generates a UI event, which is handled as an internal agent message. This causes the agent to execute the *text_out_s* script, which causes a speak event to be issued through the *speak_event* tuple. The TextAreaAgent of each user in the system will pick up the speak event through the query defined in lines 7-14, which reacts to the event by adding its contents to the text displayed in its text area.

7.5 VETk Data Constraint language

VDC (Vetk Data Constraint language) is meant for the specification of constraints on the shared data of an application. VDC enables the integrated use of ERDs, state

automata, and logic specifications. The constraints currently supported are global invariants. Often, you'd want to specify state constraints before or after certain actions (pre- and postconditions), but these are as yet not easily supported.

We have already discussed the combination of structure specifications with logic, which is a natural way to enable the logic assertions to refer to the necessary information. We use an OCL-like logic assertion language. Of the languages we discussed, OCL was deemed the most interesting for our purpose, as it combines support for sets, quantifiers, and data structure queries with assertion checking capabilities. OCL's navigational expressions are replaced by the more full-featured database queries already supported in our glue language.

We will integrate state automata in this specification as well. We adopt the UML (Object Management Group, 2001) practice of having a state automaton correspond to an entity (an object in UML). When an entity of a specific type is created, a state automaton is also created that is initialised to its start state. We adopt the statechart practice of specifying state transitions by means of an event / condition pair, with both the event and condition specified as truth expressions in the logic assertion language. This provides a suitably powerful coupling of the state automaton with the rest of the system, although this precludes standard approaches to model checking of an ensemble of such automata, since they interact through a complex data structure. This however provide the extra coupling of the automaton specifications with the rest of the system, which we found lacking in process algebras.

With this scheme, we in fact provide an implementation of the plans for UML 2.0 to formalise structure diagrams and statecharts, and integrate them with OCL. While VDC is a textual language, we provide diagrammatic versions of the ERDs and state automata by the relatively simple scheme of generating the diagrams from the textual code, with their layout generated automatically. In other words, the diagrams are view specifications. While a graphical editor may have advantages, graphical editing is sometimes problematic, and the editor needs to be good to give this method an advantage, which is not an easy thing to ensure.

We integrate VDC specifications by means of assertion checking, which we already identified as a good way to verify executable specifications. Assertion checking is typically done by adding some checking code at particular points in the program, which does nothing but check the state of some variables. In our system, assertion checking is done by generating a special agent from the assertion specifications. The agent views the necessary database information, and generates diagnostic output.

VDC specifications are compiled to generate an assertion checking agent containing all correctness constraints, an ERD, and a state diagram for each state automaton. Diagram layout is performed automatically with help of the Visualisation of Compiler Graphs (VCG) tool. While not ideal, VCG is available on most platforms and does a good job on state automata and a reasonable job on ERDs. Different ERD layouts may be generated by re-compiling, which causes the ERD to be re-generated with random start parameters.

7.5.1 Language structure

VDC is based on VETkScript, and uses some of the same syntax. VDC supports C-like comments (/* */ and //).

Queries may be specified, using almost the same syntax as VETkScript queries. The difference is that, for each query, the query result (also called a *set*) is given a name, which may be used in assertions and state transitions elsewhere in the specification. Ex. *set positions(~AGENT,~X,~Y) : position(~AGENT,~X,~Y)* defines a set of 3-tuples called *positions*, defined as the set of *position* tuples found in the system.

Sets may be defined in two places: separately, denoting a global set, and inside an entity declaration, indicating a set that is local to that entity. In local sets, we may use the identifier *self* to denote the entity that the set is specified in, in order to define sets relative to that entity.

The language enables specification of an entity-relationship model. One may define entities and relationships by means of **entity** <**name**>**(** <**parameters**> **)** { <**body**> } and **relation** <**name**>**(** <**parameters**> **)** { <**body**> } clauses. The list of *parameters* may contain both attributes and references to entities with specified multiplicities. The entities and relations are mapped to database tuples in a straightforward manner, with the first tuple argument being the entity ID in case of entities.

Ex. *entity position(x, y)* { } specifies an entity which translates to the tuple *position(ID, x, y)*.

Ex. *relation selected(user:*,item:0..1)* { } specifies a relation between users and items, with each user being related to zero or one item (multiplicity 0..1), and each item being related to any number of users (multiplicity *, or 0..∞). It translates to the tuple *selected(UserID, ItemID)*.

An entity or relation has a body, which specifies values (which are optional attributes), events, and sets. Values and events are denoted by an identifier, followed by an optional multiplicity. These translate to separate tuples with the entity ID as the first argument.

Ex. *value status;* translates to a tuple *status(EntityID, value)*. *value isjoined/0* translates to a tuple *isjoined(EntityID)*. Events are analogous.

We may specify state machines inside entities. These are basic finite-state automatons, with the difference that a transition may be specified by two arbitrary assertions: one specifies an event, and the other a state condition. A transition occurs when the state condition is true and the event occurs (i.e. the condition specified by its assertion goes from false to true). Example:

```
statemachine {
    BeginState -> SecondState {<event_assertion1>}
    SecondState -> BeginState {} {<condition_assertion1>}
                -> FinalState {<event2>} {<condition2>}
}
```

The very first source state specified is the begin state, which is entered as soon as the entity is created. If the source state of a transition is left out, the same sourcestate as

the previous transition is assumed. We may leave out either the event or condition.

Assertions may be specified inside an entity's or relation's body, in which case they are checked for each entity/relation separately, or as separate statements, indicating that they are global. In case the assertion is local, we may specify an automaton state. The assertion only has to hold when the entity is in that state.

Ex. *assert BeginState,SecondState* { <*assertion1*> } : *assertion1* has to hold in either of the two specified states.

Left to explain are the events, conditions, and assertions. These are OCL-like truth expressions. This includes Boolean expressions which specify some properties of sets, which may be combined by Boolean operators, such as **and, or, not, implies**. The Boolean expressions are quantifiers and a size operator for counting the number of elements in a set. They take as argument a *set expression*, which is any VETkScript expression in which the individual fields from the sets may be addressed as variables. The expression is expanded to form a set of expressions, one for every combination of fields from each of the sets.

Ex. *exists(position.x == 0)* is true when at least one of the elements in the set *position* has its *x* field equal to zero. *exists(set1.field1 == set2.field2)* is true when there is at least one *field1* in *set1* that is equal to a *field2* in *set2*. *size (set3) > 1 and forall (set3.field3 > 0)* is true when there is at least one element in *set3* and all of the elements of *set3* have their *field3* greater than zero.

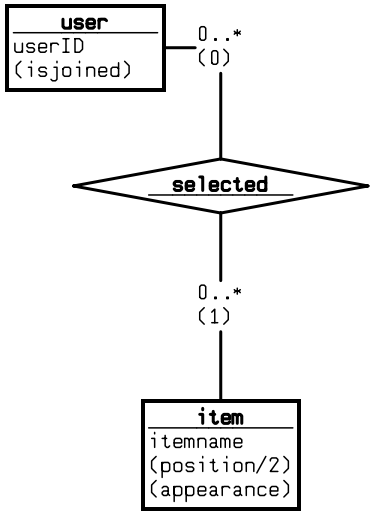
7.5.2 Example

This example specifies a minimal applications in which there are items (such as graphical items on a canvas) which users may select and operate on. Items and users are entities, and item selection is modelled by a *selected* relation. A user has a status, which indicates whether it is joined in the application or is temporarily away. An item has a name, a position, and an appearance. We specify a few illustrative example constraints, such as that a 'restricted' user may not select more than 10 items, and that a 'forbidden' user may not select items.

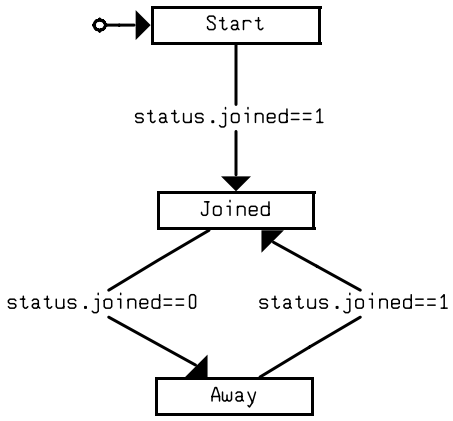
```
1  entity user(userID) {
2      value isjoined;
3      /* the selections that this user made: */
4      set myselections(~item) : selected(self, ~item);
5      /* the status of this user: */
6      set status(~userID,~joined) :
7          user(self,~userID) isjoined(self, ~joined);
8      /* a state automaton modelling the transitions of the user status: */
9      statemachine {
10         Start -> Joined {status.joined==1}
11         Joined -> Away {status.joined==0}
12         Away -> Joined {status.joined==1}
13     }
14     /* in the Start and Away state, the user may not have selected items,
```

```
15        * i.e. myselections is empty: */
16       assert Start {not myselections}
17       assert Away  {not myselections}
18       /* A RestrictedUser may not select more than 10 items: */
19       assert { not status.userID == RestrictedUser
20                    or size(myselections) <= 10 }
21 }
22 entity item(itemname)                  {  value position/2, appearance;  }
23 relation selected(user:*,item:*) { }
24
25 /* A user ForbiddenUser may not select items: */
26 set userselected(~userID, ~item) :
27       selected(~userid, ~item) user(~userid, ~userID);
28 assert{forall(userselected.userID != ForbiddenUser)}
```

Below, we explain the ERD and state diagram that are generated from the specification.



The boxes are entities. The rhomb shapes are relations. The arcs are annotated with the relational multiplicities (plus a number indicating the reference's parameter number, which is only useful for implementation purposes). Bracketed attributes indicate optional values. Bracketed and underlined attributes indicate events. The numbers following the '/' after these attributes indicate the number of arguments, i.e. '/0' means the attribute indicates just a signal without content, '/2', a 2-tuple. The default is '/1'. References from entities to entities are indicated by arcs with arrows pointing to the destination of the reference.



The boxes indicate states, the arcs indicate transitions which are labeled by events (denoted as truth expressions) and/or conditions (denoted by truth expressions between square brackets). The horizontal arrow on the left of a state indicates the begin state.

7.6 VETk Component Language

An agent is specified in VETk Component Language (VCL). VCL is a language that integrates VETk agents with the Java language. It provides algorithms and software libraries that comprise the basic functionality of an agent. Each agent is an instance of a Java class. Agents may quite easily incorporate arbitrary Java class libraries. Though agents may be written directly in Java, VCL is much more concise. A VCL agent specification compiles to a Java class.

VCL has a dual purpose: specification of the agent interface, and of the agent's overall behaviour or control flow. It specifies the input and output points of an agent, and how the agent reacts to each kind of input and how and when it generates output. VCL is tightly coupled with Java, and enables Java objects and statements in its behaviour specifications. In place of VCL, other languages might be used to specify behaviour, such as a production rule language to specify intelligent behaviour. How precisely this should be done is a subject of future research.

The definition of an input point, also called *input handler*, defines a database view. A database view consists of a set of tuples representing the current database result. All database views are accessible from anywhere in the agent's body. Any new results from the database are passed to the agent in the form of updates to the appropriate views. The updates occur in the form of additions and deletions to the view's set of tuples. In addition to creating a view, the input handler may define some code which is executed for each update, so that the agent may react to changes appropriately. While all input handlers imply views, it is sometimes useful to define an input handler without actually using its view. In this case, an update arriving at the handler can be used as a regular method call, with the handler's code being the method. The event update (@update . and @updateq .) are specially meant for this, i.e. they can be used to emulate method

calls. A future version may incorporate a separate type of input handler to signify regular method definitions.

A VCL specification can be made to remain concise, as one is able to move complex algorithms towards regular Java classes, so that only overall control flow and behaviour remains in the specification. In the ideal case that only standard Java classes known to the developer are used, the VCL specification is a self-contained reference documentation of the agent's behaviour, sufficient to understand the agent well enough to employ it in a system.

It should eventually be possible to generate interface specifications from VCL code, that is, specifications of the agent's inputs and outputs only, so that the necessary syntactical detail can be easily looked up. This could be done using a Javadoc-type approach. The resulting specifications should be easy to browse (for example by using HTML), making it even easier to select and use agents.

7.6.1 Structure

A VCL specification consists of zero or more agent specifications. An agent is specified by its name and the arguments it takes at startup, and its body:

```
'agent' <name> '(' <argument> ',' <argument ','  ...  ')'
'{'
    <agent_body>
'}'
```

Each argument specifies a Java object type, and may be assigned a default value when the argument is not set on initialisation. All arguments may be accessed from anywhere within the agent's body.

The arguments include 'regular' arguments as well as scripts. In the current version, scripts are simply arguments of type String. The variable IDs of scripts are usually suffixed by a '_s' (i.e. a script named myscript is typically called myscript_s). In future versions of VCL, scripts should be assigned a proper type and enable the parameters it expects to be defined.

An agent has several implicit arguments, in particular an initialisation script *init_s* and its window width and height.

The agent's body contains a number of clauses. Each clause has a body containing regular Java code, with exception of the **imports** clause, which contains a list of classes to import. Here is a list of the different clause types:

- **inline** - code to be included literally in the Java class to be generated. This can be used to define variables or regular internal methods.
- **init** - code to execute when agent initialises
- **startupdate** - code to execute at the beginning of an update (consisting of one or more elements of one or more views to be added or deleted).

- **endupdate** - code to execute when all elements of the update have been handled.
- **'handle'** <**view_id**> **'('** <**parameters**> **')'** - Define a database view *view_id*, and an input handler, which is the code to execute as reaction to an addition or deletion of the view's current result. Within the code, the booleans *is_addition* and *is_deletion* may be used to determine the nature of the update.
- **'handleinternal'** <**handler_id**> **'('** <**parameters**> **')'** - code to execute when an internal message arrives.

7.6.2 Functions

There are a couple of special language constructs, and several methods that can be called from within a clause body. The language constructs are meant for processing the elements of a view. There is the **foreach** statement, that enumerates the elements of a view, and the **take** statement, that takes an arbitrary element from a view.

Another way to process results is by obtaining the set of results by means of the method:

- **Results results(String view_id)** - Retrieve the current content of the given view.

Results is a Java type that can be enumerated. Another important method is the **execscript** method:

- **void execscript(String scr [, String id1,Object value1 [, String id2,Object value2 [, String id3,Object value3]]])** - Assign values to the script variables named id1,id2,..., then execute script scr.

Then there is the **exit** method for deleting an agent.

- **void exit(...)** - Exit silently if no argument is given, and with error if a String argument is given.

7.6.3 Example

We will give an example of a GUI list box agent, which is basically a wrapper around the Java AWT List class. A list of items is displayed, and items in the list can be selected. The list can be defined by means of the *item_list* view, and is dynamically updated to reflect the contents of this view. When an item is selected, the *selected_s* script is executed, with as parameter the name of the selected item. The list select event is an internal event, generated by the Java GUI event handler thread. A special event listener *AWTEventGen* is used to redirect the event to the internal action handler *select*.

...
... **... ... ...**
...

```
107agent ListboxAgent(Integer rows=new Integer(5),
108String selected_s /* selected_s(String item) */
109) {
110     inline {
111          /* code inserted literally in the agent, defining the variables */
112          List list;
113          Hashtable items=new Hashtable();
114     }
115     init {
116          /* initialisation code: create the listbox and listen to it */
117          list=new List(rows.intValue());
118          add(list);
119          list.addKeyListener(this);
120          validate();
121          list.addActionListener(new AWTEventGen(self,"select"));
122     }
123     /* Handles events emitted by the listbox */
124     handleinternal select(ActionEvent e) {
125          execscript(selected_s,"item",items.get(e.getActionCommand()));
126     }
127     /* the list of items to display in the box */
128     handle item_list(Object key,Object value) {
129          if (is_addition) {
130               /* add item */
131               items.put(key.toString(),value);
132               list.add(key.toString());
133          } else if (is_deletion) {
134               /* delete item */
135               items.remove(key.toString());
136               try {
137                    list.remove(key.toString());
138               } catch (IllegalArgumentException e) { e.printStackTrace(); }
139          }
140     }
141}
...
...          ... ... ...
...
350
```

Note that the names and types of the output scripts' parameters is defined inside comments. In future versions of VCL, there should be facilities to define them using language constructs.

7.7 Conclusion

In this chapter, we defined the languages and execution model of the VETk software toolkit, and its rationale and design choices, linking them to our findings in the previous

chapters. This provides the necessary information for the next chapter, chapter 8, in which we will discuss methodology and some examples at length.

Along the chapter, we mentioned several weaknesses that were noted as candidates for future improvements. To conclude this chapter, we will summarise these.

- The database does not define access policies. Since some applications are based on exclusive write rights of some agents to some parts of the data, access policies may formally ensure that this is indeed the case.
- No specification of script parameters, scripts are just Strings. Script parameters will enable script execution to be type checked.
- A special construct for a regular function call, rather than specifying a function call as a kind of view handler. While this facility will not have great impact upon conciseness, it will enable the designers to better specify their intentions.
- The HTML applet tags are a bit verbose, and possibly, a separate front-end language (similar to PHP) may be used to generate these.
- Applets cannot define their own window size, which is very inconvenient, as sizes sometimes have to matched in several places. This is mainly a weakness of existing browsers.
- Generation of interface specifications (automatically generated documentation, a la Javadoc) from VCL would be useful.
- An exit_s script should be defined. While the default exit behaviour is usually acceptable, this provides an easy way to handle non-default exit behaviour.
- An agent-delete command should be defined, as a better replacement than the current method of closing an agent's window. This enables agents to delete each other more easily.
- One should be able to specify pre- and postconditions in VDC. These could be specified as assertions in VETkScript code.
- The database query facility is lacking a *not* operator.

Chapter 8

Using the technique

8.1 Introduction

In this chapter, we will illustrate and discuss the technique. We will do this by means of preliminary development guidelines and methodology, and several moderate-size example programs, totalling about 100-500 lines of code each. As we have argued, giving ample examples is a good way to illustrate a specification technique. Each example illustrates some of the ideas of the technique and ways in which it can be used.

Of each example, we will give a natural language specification, a screen layout, the VDC specification with corresponding diagrams, and some particularly interesting sections of the VCL, VETkScript, or HTML specifications. The specifications will be well commented, so that the reader who is unfamiliar with the specification languages will be led through the specifications more easily.

With help of these specifications, we will provide a more in-depth assessment of our technique. We will look at whether our technique fulfills our expectations, and will mention both small and large problems with the current version of the technique. We will also look at the distinction between 'domain data' and 'user interface data' as identified earlier. At the end of each example, we will give a short summary of the most important conclusions, separated into: advantages (what has been successfully achieved with help of our technique), problems (things that should be improved), and issues (things that are open problems, which may or may not be directly related to the technique). We will conclude the chapter with some general findings.

We will give four example applications: simplechat, a chat room, with a help agent added to it, illustrating a simple application and the specification of a help agent for an application; a shared blackboard, indicating how graphical objects on a canvas and users operating on them may be modelled; mini VMC, a version of the running example, to illustrate modelling of web browsing, a virtual world, and a multimodal dialogue agent; Webset, a card game that illustrates the use of a domain agent that manages the

game rules.

8.2 Using the technique

8.2.1 Modelling a system as entities and agents

As we have mentioned, entities correspond directly to software components. Components play a similar role to components and objects found in regular programming languages and GUI toolkits, with each GUI or VE object being modelled as a component. When a component is created, it is assigned a unique entity ID that it may use to read and write the shared database. While it is an obvious design choice to have a component operate only on its own entity and its relations, the entity ID is only provided as a convenience, and a component retains complete freedom in how it uses the database. We see the specification of access policies to clarify purpose and increase safety of database access as a separate problem, which is left as a topic of future research. Not all components need to have entities, nor do all entities need to have components, and there is no correspondence required between component types and entity types. This enables some freedom in modelling the abstract application structure and its implementation by means of agents.

Note that we have mentioned the classification of data into domain data and UI data. We encourage modelling both types of data in our shared database. In some cases, separation of domain and UI data is not even very clear. User interface elements often have a relation to specific domain elements, and using a single data structure enables the explicit modelling of this relation. For example, in a graphical editor, the graphical object being edited (circles, lines, etc.) are domain data. Selection of a graphical object on a canvas may now be modelled by means of a relation between the canvas (a user interface element) and the graphical object (a domain element).

It is not necessary nor convenient to model each little button or text field as a separate entity, unless it plays a special structural role. For example, a GUI window may be effectively modelled by means of a single entity. The components in the window (buttons, etc.) may all write their information to this entity. An example of a special structural role is the hyperlink scheme we used in one of our example applications. Here, hyperlinks are modelled as button agents embedded the HTML code, rather than regular HTML hyperlinks. These agents can perform arbitrary actions if a user moves over them or clicks them. By modelling these as separate entities related to the current window displaying the webpage, it was very easy to query what hyperlinks the user was viewing and selecting.

The world structure of a VE may be effectively modelled as an E/R structure, with for example rooms and objects being entities, and containment of an object inside a room being a relation. This enables an agent to query the VE world easily. Note that this is another interesting mixture of domain data with user interface data, as the world may be seen as domain data, while some of the objects inside it are part of the user interface.

When the domain data is more than simple, static data that can be edited directly by

the user interface components, but requires some management, we may introduce one or more domain agents that operate on the domain data. The user interface agents only generate requests to these agents to make the desired domain data changes. These agents play the role of Model components in MVC terminology.

8.2.2 A preliminary methodology

Figure 8.1 gives a rough idea of how the technique may be used in a development process. One starts development with some informal specifications. In chapter 3 we have seen that these typically include screen layouts and/or some manner of interface structure specifications, event sequencing or dialogue flow, and description of some of the interactive objects. These may be translated to an ERD and some state automata that may be incorporated in a VDC requirements specification. As soon as the ERD is known, we may also start building some graphical layouts with some basic behaviour using the glue languages, as long as only standard components are used. While developing a program, the ERD alone should provide a good reference for seeing where one can find and store one's data, while the other data constraint specification serve as sanity checks. Thus we obtain a first prototype that may be evaluated with users if necessary. Meanwhile, we may specify the custom agents (such as interface agents and other special interactive objects, and domain functionality). After integrating these custom agents, we obtain a fully functional prototype, that may eventually be developed into the final system by means of incremental changes.

8.3 Example specifications

8.4 Simplechat: a chat room with help agent

We will start with a relatively simple VE-like application: a chat room. On the internet, Web-based chat rooms have been proliferating lately. They are typically written in Java or Javascript, combined with software running on the Web server. They are typically not particularly complex or fanciful, in fact, some are rather primitive. Such an application can easily be written in VETk using only about 100 lines of HTML code. Our chat room also includes a help agent which can be brought up to provide contextual help by observing the user's activities.

Since this is the smallest application of the four, we decided to give the full specification here. This should give an appropriate idea of what a VETk application looks like, since it includes specifications in all four languages. This application illustrates the VE properties: graphical interface, multi-user, and interface agents.

In simplechat, users can enter the chat room by loading the appropriate HTML page. This brings up a screen with a text area displaying the chat messages, a text field where messages can be typed, and some controls for entering a user name and selecting a different action. There are three kinds of actions: talk, which means sending messages to everyone in the room, send private message, which means talking to one specific

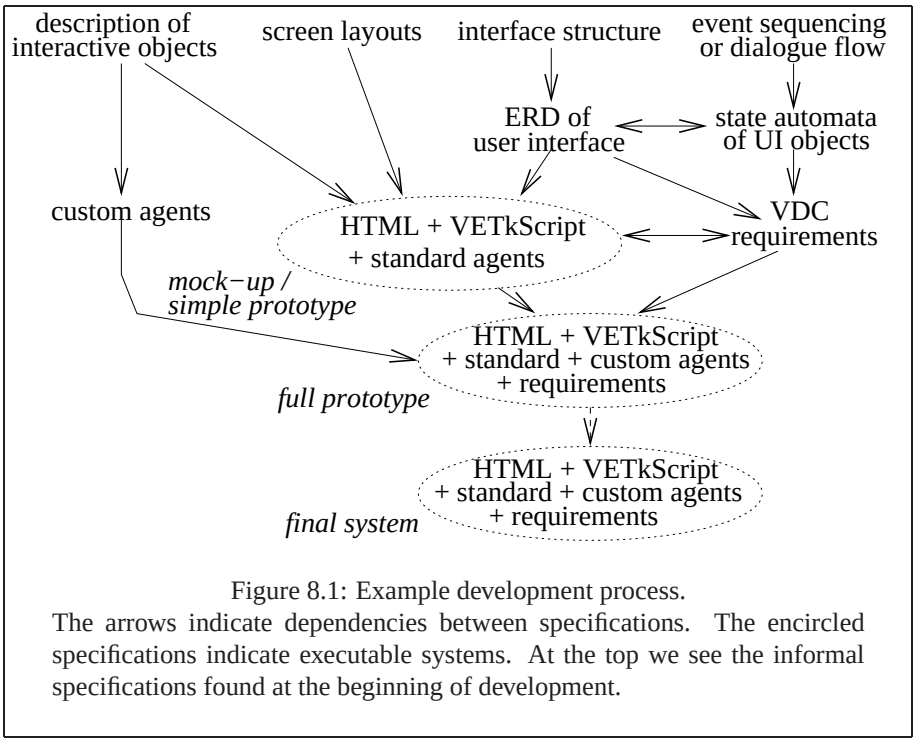
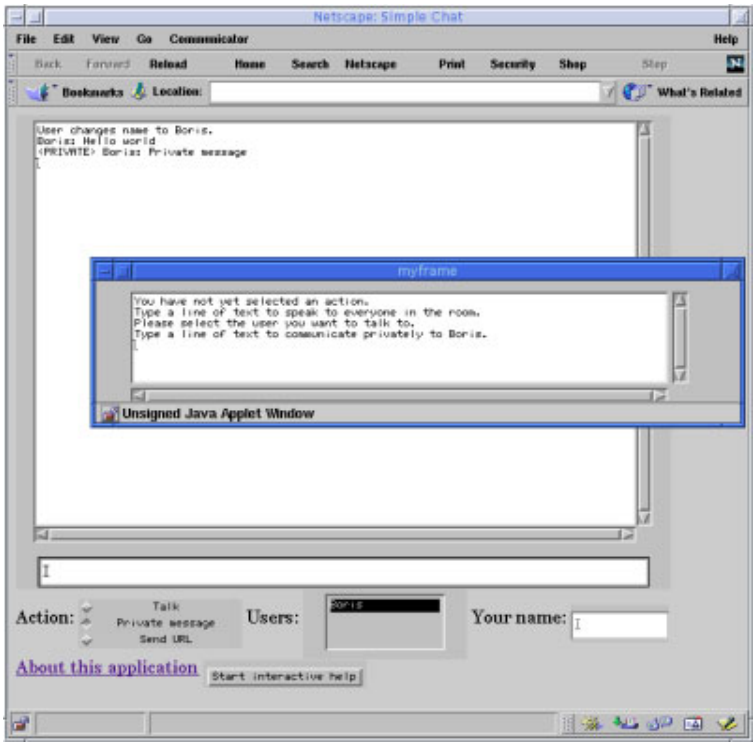


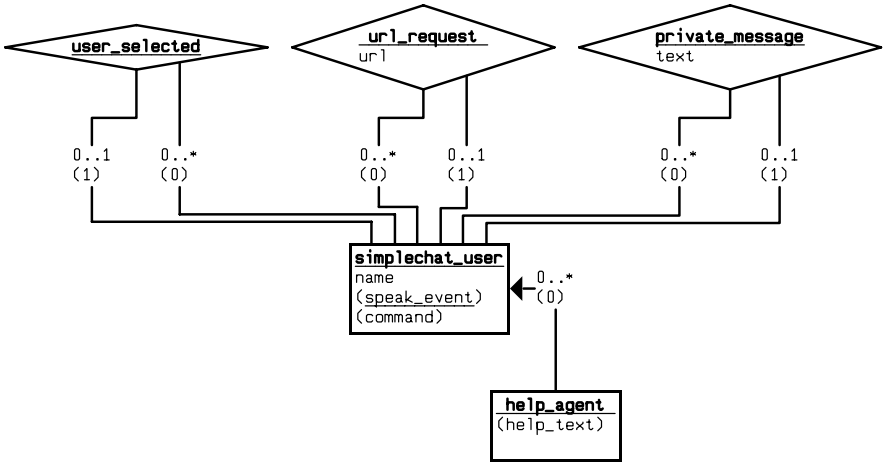
Figure 8.1: Example development process.
The arrows indicate dependencies between specifications. The encircled specifications indicate executable systems. At the top we see the informal specifications found at the beginning of development.

user, and send URL, which means displaying a given Web page in a specific user's Web browser. The users are identified by names which are displayed in front of their messages, and there is a list of the users currently online, which can also be used to select a specific user. This application also has a help agent, which can be brought up when the user desires it. The help agent gives contextual help, telling whether the user should enter his/her name, select an action, a user, and what the currently selected action does.



8.4.1 Structure

The application's structure is given by the ERD below.



It is an obvious design choice to model a user by an entity, *simplechat_user*. The three different types of actions talk, private message, and send URL, are modelled by

a *speak_event* event attribute, and the two relations *private_message* and *url_request* respectively. The latter two actions are naturally modelled as relations, as they relate to a specific user for which the actions are meant. All three signal events, that is, their creation indicates that a particular event occurs, which may be processed appropriately. After creation, they are immediately deleted again. Events are similar in concept to procedure calls.

There are two more user interface state properties that we find in our application: the currently selected command and the currently selected user, which are used to determine what to do when the user issues a line of text. They are modelled by the value attribute *command* and the relation *user_selected*, respectively. While it is quite easy to show this user interface state information to other users, it is not very meaningful to do this, and the information was at first useful for the respective user only. When we decided to add the help agent however, the help agent also used this information, which could be obtained in a comprehensive way, and without changing the original application. This little story shows that information that is first considered local, is later promoted to global use. This illustrates one of the aforementioned advantages of our technique over more traditional encapsulation approaches. In more traditional approaches, such local data would for example have been encapsulated as an arbitrarily-modelled bunch of local variables within a component *simplechat_user*. The need to communicate this data to the help agent would have required some redesign of this component, consisting of finding out where the particular variables needed are found, adding some functions to access this data, and defining some connection to the *simplechat_user* and the *help_agent* components. In our technique, all we need to do is take a look at the ERD to see where we can find the information we need for our new agent.

Note that we do not specify any access policies, while this may be desirable in many cases. For example, we may state that certain agents may not modify certain data, which enables agents to make certain assumptions about the changes made to the data they are working on. Access policy specification, which is probably most naturally specified in either glue specifications or VDC specifications, is a subject of future research. One could opt for a scheme similar to those found in object- and component-based languages, but something more is probably needed, in particular the ability to set read and write permissions separately, which would also have been useful for object-oriented languages.

A more complete data specification is found in the VDC specification below.

```
1  entity simplechat_user(name) {
2      // Message to be spoken to everyone in the room
3      event speak_event;
4      // The currently selected command
5      value command;
6      set sendsprivate(~user,~text) : private_message(self,~user,~text);
7      set sendsurl(~user,~url)      : url_request(self,~user,~url);
8      // state machine describing the possible actions. Specifies that the
9      // private_message and url_request are in fact events.
10     statemachine {
11         Quiet -> Speak      {} {speak_event}
```

```
12                  -> Private   {} {sendsprivate}
13                  -> ShowsURL  {} {sendsurl}
14          Speak      -> Quiet  {} {not speak_event}
15          Private   -> Quiet  {} {not sendsprivate}
16          ShowsURL -> Quiet  {} {not sendsurl}
17      }
18      // state constraints: make sure no two actions occur at the same time.
19      assert Speak      {not sendsprivate and not sendsurl}
20      assert Private   {not speak_event   and not sendsurl}
21      assert ShowsURL {not speak_event   and not sendsprivate}
22      // value range specification
23      assert {not command or command.val=="None" or command.val=="talk"
24                or command.val=="private" or command.val=="sendurl"}
25  }
26  // the currently selected user
27  relation user_selected(simplechat_user:*, simplechat_user:0..1) {}
28  // These two relations are actually events, in that they signal the actions
29  // they stand for by their creation and subsequent deletion.
30  relation private_message(simplechat_user:*, simplechat_user:0..1, text) {}
31  relation url_request(simplechat_user:*, simplechat_user:0..1, url) {}
32  // the help agent, giving help to a specific user, through setting a textual
33  // hint in help_text
34  entity help_agent(simplechat_user:*) {
35      value help_text;
36  }
```

In addition to the ERD, this specification models a state machine (lines 10-17) that models the sequencing of the relations and events concerned with the user's communication actions. In particular, it asserts that each action is done by creation and subsequent deletion of the appropriate event or relation. The state machine is modelled by reacting on the presence or absence of elements in the three sets *speak_event* (corresponding to the value attribute with the same name), and *sendsprivate* and *sendsurl*, as defined in line 6-7. Some assertions (line 18-21) now assert that only one such event may happen at a time.

The state automaton specification is suitably readable, and we decided not to include the generated diagram here. The transition conditions can be specified quite concisely. The fact that we defined the appropriate sets separately, rather than inside of the transitions (as with OCL), helps a great deal here.

One more assertion (line 22-24) asserts the range of values that a command may have. While we cannot really verify if this specification specifies the complete set of constraints that the data structure should conform to, intuitively we have tried to be complete. We did not omit anything that did come to mind but turned out to be unspecifiable. Yet, the specification is concise and should be quite readable, especially with help of the generated diagrams. This means that this kind of specification is suitable to specify the appropriate dat constraints.

8.4.2 Main application

The main application consists of HTML code, using standard agents with mostly straightforward glue code. With help of the comments, the code should be straightforward to understand.

```
<HTML><HEAD> <TITLE>Simple Chat</TITLE> </HEAD><BODY>
<!– Displays what happens in the chat room –>
<applet code="TextAreaAgent.class" width=600 height=400 align="middle">
<param name="columns" value="90"> <param name="rows" value="30">
<param name="init_s" value='
    /* A user speaks. When a speak_event occurs, the append handler makes
     * sure that its content and the user who issues the event are appended to
     * the textarea's text (i.e. displayed). As mentioned before, the append
     * handler does not maintain a view, but just processes the additions. */
    @addquery append ~NAME, ": ", ~TEXT,"\\n" :
        simplechat_user(~USER, ~NAME) speak_event(~USER, ~TEXT);
    /* A user changes name. When simplechat_user is redefined, this results
     * in an addition which results in the display of this text message. */
    @addquery append "User is now called: "+~NAME+".\\n" :
        simplechat_user(~USER, ~NAME);
    /* A private message. */
    @addquery append "<PRIVATE> ",~NAME,": ",~TEXT,"\\n" :
        simplechat_user(~USER, ~NAME) private_message(user_agent,~USER,~TEXT);
'></applet>
<BR>
<!– user may type text to be communicated here –>
<applet code="TextFieldAgent.class" width=600 height=35 align="middle">
<param name="columns" value="90">
<param name="text_out_s" value='
    /* issue text according to the current command */
    /* get the current command and the user's selection */
    cmd = @doquery ~CMD : command(user_agent, ~CMD);
    selected = @doquery ~USER : selected_user(user_agent,~USER);
    /* react to the command. Note that cmd is always a set with zero or one
     * elements. */
    for x in cmd : {{ for y in x : {{
        if y == "talk" : /* talk to all others */
            @update . speak_event(user_agent, text);
        elif y == "private" and len(selected) > 0 : /* private message */
            @updateq . private_message(~USER,user_agent,text) :
                selected_user(user_agent, ~USER);
        elif y == "sendurl" and len(selected) > 0 : /* send URL */
            @updateq . url_request(~USER,self,text) :
                selected_user(user_agent, ~USER);
        else : {{ /* error */
            /* create dialog box with 'OK' button */
            @window("error_win",250,100,1,2);
            @new err_label LabelAgent
                text="Please select a command and/or user";
```

```
                @new ok_but ButtonAgent text="OK",
                    pressed_s=<<@wclose(self_window);>>;
                @wadd(error_win, err_label_window);
                @wadd(error_win, ok_but_window);
                @wadd(root,error_win);
            }}
        }} }}
'></applet>
<BR>
Action:
<!– Command selection –>
<applet code="CheckboxGroupAgent.class" width=150 height=45 align="middle">
<param name="init_s" value='
    /* define the contents of the checkbox as all checkbox tuples found on
     * this agent */
    @addquery checkbox ~LABEL, ~VAL : checkbox(self, ~LABEL, ~VAL);
    /* define these checkbox tuples */
    @update +checkbox(self, "Talk", "talk");
    @update +checkbox(self, "Private message", "private");
    @update +checkbox(self, "Send URL", "sendurl");
'><param name="selected_s" value='
    /* set the user command when the user selects an item */
    @update + command(user_agent, *value);
'></applet>
Users:
<!– user display and selection –>
<applet code="ListboxAgent.class" width=150 height=65 align="middle">
<param name="init_s" value='
    /* display the set of users in the system in the listbox */
    @addquery item_list ~K, ~V: simplechat_user(~V, ~K);
'><param name="selected_s" value='
    /* set the selected_user when the user selected an item */
    @update + selected_user(user_agent, *item);
'></applet>
Your name:
<!– The user's name can be entered here –>
<applet code="TextFieldAgent.class" width=90 height=35 align="middle">
<param name="columns" value="12">
<param name="text_out_s" value='
    /* Overwrite the user's name when text is entered here */
    @update + simplechat_user(user_agent, *text);
'>
</applet>
<BR>
<!– Textual help –>
<a href="simplechat_help.html">About this application</a>
<!– Press button to activate help agent –>
<applet code="ButtonAgent.class" width=150 height=25 align="middle">
<param name="text" value="Start interactive help">
<param name="pressed_s" value='
```

```
95      /* start help agent, passing the user's entity through the originator
96        * variable */
97      originator=user_agent;
98      @execute("vs","simplechat_help.vs");
99  '></applet>
100<!– agent that processes show_document requests from outside, its entity
101       stores the user information –>
102<applet code="WebpageAgent.class" width=0 height=0 align="middle"
103name="user_agent">
104<param name="init_s" value='
105     /* process show_document requests */
106     @addquery show_document ~URL," blank" : url_request(self,_,~URL);
107     /* define the user's attributes */
108     @update + simplechat_user(self, *"Anonymous");
109     @update + command(self,"None");
110'></applet>
111</BODY></HTML>
```

The specification uses standard agents only (with exception of the help agent), and consists mostly of queries and updates. The only place where some explicit control flow occurs in in the processing of the text message (lines 23-45), where a series of *if*s decide what happens with the typed text. The main reason for this is that the display of the error message in a separate window (lines 36-43) could not be done using a *@updateq* statement. This means that the glue language is in this case appropriately powerful to concisely glue generic agents into a specific application. The agents are mostly traditional user interface components, inherited in this case from Java GUI toolkit, with the exception of a special agent *WebpageAgent* to handle external Web page display requests (line 90).

It was straightforward to produce the simple visual layout of this application using HTML tags. The standard layout scheme used here, having a number of rows of objects, can be achieved easily by the standard HTML text flow mechanism, combined with <*BR*>s to indicate a row is full and the following objects should be displayed on the next row. Text and even hyperlinks leading to help information can be added naturally, without having to create label objects to hold text, or do more complicated things to produce links and nicely-formatted text. Visual layout maps to the lexical order in which the components can be specified: the order of the components is the same as the order in which they occur in the layout. This mapping would have been perfectly intuitive if we had used some kind of visual language, but here, some translation step is required. Since this specification is short (1.5 page), as it is supposed to be, this should not be a problem.

The ensemble of agents store their information in a single entity, represented by the *WebpageAgent*. The choice is arbitrary, it might have been any other agent on the webpage. Because the applets have access to the applet IDs through script variables, it is quite easy to use the appropriate entity.

There is one particular point in which this application is lacking due to difficulty of specification. If a user changes his/her name, we wished to produce a message like:

'user changes name from X to Y'. However, this would require the old value of the user name to be accessible through a query, so this was not possible. We would have to write much lengthier specifications or even add data to our data model to remedy this problem.

8.4.3 Help agent

The help agent is a separate part of the software, and was added later. No changes needed to be made to the application to add this agent, except that a button was added that starts up the agent. The agent itself, written in VCL, is given below.

```
1  agent SimpleChatHelpAgent(String noname, String output_s) {
2      inline {
3          /* The different action types */
4          static final int NONE=0;
5          static final int TALK=1;
6          static final int PRIVATE=2;
7          static final int SENDURL=3;
8          boolean do_name_help=false, do_action_help=false;
9          /* show help on the currently selected action */
10         void action_help() {
11             take (Integer action) results("action_selected");
12             if (action==null) action=new Integer(NONE);
13             take (String user) results("user_selected");
14             switch (action.intValue()) {
15             case NONE:
16                 execscript(output_s,"text",
17                     "You have not yet selected an action.");
18             break; case TALK:
...
...        ... print more help texts according to user and action ...
...
38             }
39         }
40         /* show help on filling in the user name */
41         void name_help() {
42             take (String name) results("name_filled_in");
43             if (name!=null && !name.equals(noname)) {
44                 execscript(output_s,"text", "You are now known as "+name+".");
45             } else {
46                 execscript(output_s,"text","Please fill in your name first.");
47             }
48         }
49     }
50     endupdate {
51         /* print appropriate help texts after state changed */
52         if (do_name_help)   { name_help();   do_name_help=false; }
53         if (do_action_help) { action_help(); do_action_help=false; }
```

```
54      }
55      /* user filled in a new name */
56      handle name_filled_in(String name)       { do_name_help=true; }
57      /* user selected a new action */
58      handle action_selected(Integer action) { do_action_help=true; }
59      /* user selected a user */
60      handle user_selected(String user)        { do_action_help=true; }
61      /* exit if user goes offline */
62      handle user_is_online() {
63          if (results("user_is_online").isEmpty()) exit();
64      }
65  }
```

In this agent, the set update handling is nothing more than remembering what sets have changed, and at the end of the update, printing the appropriate help messages (*action_help* and *name_help*).

We can see here that the handling of the set updates and the execution of scripts is similar in conciseness to regular method handling and method calling in for example Java, though set updates are more powerful. This agent needs only zero- or one-element sets. It uses *take* (i.e. take an arbitrary element from a set or return null if not available) to check if the set has an element, and get the element. This statement is specially meant for this purpose. Possibly, such special cases of sets (zero or one element) may also be specified as a property of the handler.

In this and some other agent specifications, there is some trouble with choosing between triggering on additions to a set, or to deletions or the set becoming empty. In this agent, emptiness of the *user_is_online* set triggers agent exit. If we wanted the exit trigger on the fact that the set was not empty, we would have to change the agent's specification, rather than doing this by just specifying a different query. This problem has to do with the limited expressiveness of queries, which do not have a *not* operator, and which only allow set elements to be post-processed piecewise, instead of being able to form an arbitrary new set out of the query results. So, an increase of the expressiveness of queries would be able to solve this problem. Another instance where this occurs is in the main application (line 13), where a deletion triggers the display of a message. In this case, a separate handler *append_deletion* is used, which is defined for this purpose.

Finally, we give the script that starts up the help agent, which is executed when the 'start interactive help' button is pressed (lines 82-83).

```
1   // Script to start up the help agent.
2   // INPUT: originator is the user that started this script.
3   // Area where agent displays its help text
4   @new out TextAreaAgent columns=80,rows=6,width=490,height=50,
5       init_s = <<
6           @addquery append ~TEXT : help_text(self,~TEXT);
7       >>;
8   // the help agent
9   @new help SimpleChatHelpAgent noname="Anonymous",
10      init_s = <<
```

```
11          // see if user filled in name
12          @addquery name_filled_in ~NAME : help_agent(self,~USER)
13              simplechat_user(~USER,~NAME);
14          // user has not selected an action
15          @addquery action_selected 0 : help_agent(self,~USER)
16              command(~USER,"None");
...
...       ... similarly, query talk, private, and sendurl commands ...
...
26          // see if the user selected a user
27          @addquery user_selected ~NAME : help_agent(self,~USER)
28              selected_user(~USER,~SELUSER)
29              simplechat_user(~SELUSER,~NAME);
30          // see if user is still online
31          @addquery user_is_online ~USER : help_agent(self,~USER);
32      >> output_s = <<
33          text = text + '\\n';
34          @update + help_text(out,*text);
35      >>;
36  @update + help_agent(help,originator);
37  // Create and display the agent's window
38  @window("win",600,160);
39  @wadd(win,out_window);
40  @wadd(win,help_window);
41  @wadd(root,win);
```

The help agent can easily obtain its information through a number of straightforward queries. Note that, if the user exits, the help agent is deleted also, because the entity *help_agent* is deleted because it contains a reference to the user. This triggers the deletion through the *user_is_online* handler. In the current version, it is not the case that an agent is exited automatically when its entity is deleted, only vice versa. This would be a natural addition for future versions, and lines 61-65 of the help agent would not be needed.

Note that the script has an input variable, *originator*, which it inherits from the agent that started it. Unlike inputs that scripts take from agents, this input is not formally specified, and a comment is used to specify it. Explicit input specification for scripts may be added in future versions. These may also provide double-checking for inputs taken from agents, i.e. the specification of inputs could make the system check if these variables really exist.

8.4.4 Conclusions

After this first example, we may already come up with a rather large list of advantages, problems, and issues.

Advantages:

- The VDC specifications are sufficiently concise, comprising only some 36 lines

of code for the entire application.

- The glue specifications are concise. They consisted mostly of queries and updates, and little explicit control flow was needed. The distributed multi-user nature of the application did not provide any significant extra complexities.

- A simple interface agent could be added without modification of the original software, showing the concept of a shared data structure was successful in this respect.

- HTML was successful as layout specification, enabling specification of text-flow like layout, text labels, hyperlinks, and textual help.

- Separating the set specifications from the assertions was a good idea. Especially, it enables the state transitions to be specified very concisely, so that the state automaton specifications are easy to read.

- VCL may be considered sufficiently concise, as specifically the handling the sets and updates took only a few lines of code.

Disadvantages:

- As we have already indicated, it would be useful to specify access policies for the data.

- The necessity to specify the triggering on either addition or deletion from a set, or a set being either empty and non-empty, in an agent, rather than in glue code, indicates that more powerful queries are needed. For example, a *not* operator, excluding elements from a set as a result of positive matches, could be added.

- Scripts do not yet define input variables, and in some cases, this means we cannot formally specify the inputs to scripts at all.

- The old (previous) value of a tuple or the query result is not available in a query specification. In some languages, such as OCL and Z, we can access both the old and the new value in case of a change. Depending on how we should best model a ‘change’, we can create a construct for accessing previous values of a result.

- It may be useful to define a special VCL construct for zero-or-one-element set, indicating that an agent assumes a particular set satisfies this property.

- The fact that tuples may be used in the program that are not defined in the VDC specification sometimes results in erroneous use of tuples (such as updating or querying a tuple of the wrong size). This means that a checking mechanism that would disallow any tuples to be used that are not defined in the VDC specification would have detected some errors.

8.5 Blackboard: a shared blackboard

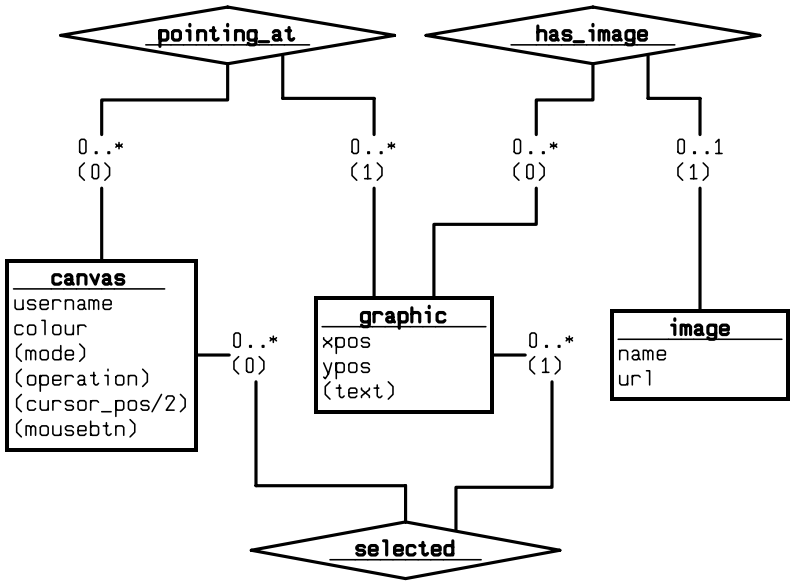
Next, we will describe a shared blackboard application. Shared blackboards, a well-known type of multi-user application, have the extra complexity of working on a graphical canvas, which may be seen as a most rudimentary form of a graphical 'virtual environment'. In this specification, we will show how such a canvas may be modelled in our technique.

Blackboard is a shared blackboard application that enables the manipulation of text objects or images. There is a canvas where the objects can be viewed and selected, a checkbox to select an operation. For enabling multi-user awareness, the user can enter his/her name and select the colour of his/her telepointer. There is a list of users and their status, and the different users' telepointers are shown on the canvas. Objects can be created by selecting them (using the left mouse button), and then moving or deleting them (using the right mouse button). Multiple objects may be selected to be operated on simultaneously. New objects may be created by selecting 'new text' or 'new image' and then clicking on the location where the object should be created. A dialog box appears in which the object can be defined. For as long as the dialog box is open, no canvas operation can be done until it is closed again (i.e. the interface is modal in this respect).

We chose to model the creation dialog by making the interface modal to show how this could be specified. While usability guidelines typically discourage modal behaviour, we do find it interesting to see how it works out, as it is likely to be harder to specify in our technique that non-modal user interfaces, because agents always continue to run in parallel. We will discuss a non-modal variant later.

8.5.1 Structure

The application's structure is given by the ERD below.



Unlike the chat room, this application has domain data: the graphical objects (*graphics*) that are being edited, and related information (*has_image* and *image*). The UI data is comprised of the *canvas* and its related information (*pointing_at* and *selected*). This is an example of the effective mixing of domain with user interface structure in a single structure specification. Since a user interface's state is likely to contain some kind of references to domain objects, this situation is likely to be normal rather than exception, and we can now make a comprehensive model of it.

The entity attributes are best explained using the VDC specification.

```
1  entity graphic(xpos,ypos) {
2      value text;
3      set is_image(~name,~url) : has_image(self,~image) image(~image,~name,~url);
4      // a graphic must at least define text or image
5      assert {text or is_image}
6  }
7  relation has_image(graphic:*,image:0..1) {}
8  entity canvas(username,colour) {
9      // mode indicates whether the user is drawing on the canvas or entering a
10     // new graphic
11     value mode;
12     // currently selected canvas operation
13     value operation;
14     // mouse state: position and button being pressed
15     value cursor_pos/2, mousebtn;
16     set cursor_pos(~x,~y) : cursor_pos(self,~x,~y);
```

```
17      // The lifecycle of the canvas' modal interface modes.
18      statemachine {
19          Initial -> Drawing   {} {mode.val=="drawing"}
20                  -> Entering {} {mode.val=="entering"}
21          Entering -> Drawing {} {mode.val=="drawing"}
22          Drawing -> Entering {} {mode.val=="entering"}
23      }
24      // cursor_pos defines the location of newly entered graphics
25      assert Entering {cursor_pos}
26      // some value range definitions
27      assert { not operation or operation.val=="move" or operation.val=="delete"
28               or operation.val=="newtext" or operation.val=="newimage" }
29      assert { not mode or mode.val=="entering" or mode.val=="drawing" }
30      assert { not mousebtn or mousebtn.val=="left" or mousebtn.val=="middle"
31               or mousebtn.val=="right" }
32      // we would like to assert that the cursor_pos does not change in Entering
33  }
34  // indicates mouse pointer points at a graphic
35  relation pointing_at(canvas:*,graphic:*) {}
36  // indicates graphic is selected for operation
37  relation selected(canvas:*,graphic:*) {}
38  // the image types available
39  entity image(name,url) {}
40  // we wish to specify that images can (as yet) not be deleted
```

Graphic has position attributes, and an optional attribute indicating that a line of text is part of its appearance. It may also have an image as part of its appearance (as referred to by *has_image*), or both, but it should have at least one of the two (lines 4-5).

Canvas has a *mode* and *operation* attribute, which indicates the mode of the modal interface, and the currently selected operation. The different values they may have are given in lines 27-29. *Mode* indicates either doing something on the canvas ("drawing") or entering a new graphic ("entering") using the current mouse cursor position to indicate the position of the graphic. This is indicated by the state automaton and the assertion that *cursor_pos* should be defined in the "entering" mode (lines 18-25).

While regular 1-argument value attributes (such as *mode*) automatically define corresponding sets with the same name, we do need to explicitly define a set for the 2-argument value *cursor_pos*. When we include names for the arguments for multi-argument sets, we may both specify this set automatically, and also specify more naturally what these arguments mean. This is an obvious candidate for future addition.

Three of the assertions specify value ranges (lines 26-31). A more concise notation for this, which is for example specified as part of the value definition, may be added in future versions.

We specify in a comment (line 40) that images should not be deleted. This is to make life easier for our application, which we will discuss below. However, we cannot formally specify this, as it means specifying dynamical behaviour, which is currently only possible by means of state automata. In future versions, we could specify this by en-

abling both the old and new state to be accessed, similar to Δ schemas in Z.

8.5.2 Main application

We continue with the main application. It consists of a single HTML specification using standard GUI agents only. We also need to define some images, which are specified in a separate script. It is quite easy to add more images later on, or to have the users define more images themselves. They will be picked up by the system, even as the user is selecting from a list of existing images.

We will only look at part of the canvas specification, and the checkbox for selecting an operation. The rest of the code is straightforward, much of it similar to what we have seen in simplechat.

```
<HTML><HEAD> <TITLE>Shared blackboard</TITLE> </HEAD><BODY>
<applet code="ObjectCanvasAgent.class" name="canvas"
width=450 height=300 align="middle">
<param name="init_s" value='
    /* init canvas entity */
    @update + canvas(self,"Anonymous","red");
    @update + mode(self,"drawing");
    @update + operation(self,"move");
    /* show graphics with text attribute */
    @addquery text_object ~OBJ, "b", ~TEXT, ~X, ~Y :
        graphic(~OBJ, ~X, ~Y) text(~OBJ, ~TEXT);
    /* show graphics which have images */
    @addquery graphical_object ~OBJ, "a", ~URL, ~X, ~Y :
        graphic(~OBJ, ~X, ~Y) has_image(~OBJ, ~IMG) image(~IMG,_,~URL);
    /* show selections */
    @addquery graphical_object ~OBJ, "d", "selected_"+~COLOUR+".gif", ~X,~Y :
        graphic(~OBJ, ~X, ~Y) selected(self, ~OBJ) canvas(self,_,~COLOUR);
    /* show the users telepointers */
    @addquery graphical_object ~OBJ, "c", "pointer_"+~COLOUR+".gif", ~X, ~Y :
        cursor_pos(~OBJ, ~X, ~Y) canvas(~OBJ,_,~COLOUR);
'><param name="pointer_moved_s" value='
    /* move operation: move the selected graphics */
    @updateq + graphic(~OBJ, *~OBJX+xpos-~OLDX, *~OBJY+ypos-~OLDY) :
        mousebtn(self,"right") mode(self,"drawing") operation(self, "move")
        graphic(~OBJ, ~OBJX, ~OBJY)
        selected(self, ~OBJ) cursor_pos(self, ~OLDX, ~OLDY);
    /* delete operation: delete the selected graphics */
    @updateq - graphic(~OBJ,_,_) text(~OBJ) has_image(~OBJ) selected(_,~OBJ) :
        mousebtn(self,"right") mode(self,"drawing") operation(self, "delete")
        selected(self, ~OBJ);
    /* only update cursor_pos in drawing mode */
    @update + drawingmode(self,"drawing");
    @updateq + cursor_pos(self, *xpos, *ypos) :
        mode(self,~MODE) drawingmode(self,~MODE);
'><param name="pointing_at_s"
```

```
36          value='@updateq state pointing_at(self, source) : graphic(source,~X,~Y);
37 '><param name="selected_s" value='
...
...         ... selected_s, creates dialogs for creating new graphic objects ...
...
84 '></applet>
85 <!– The command to select.  Note that the checkbox updates itself when the
86       command is changed elsewhere –>
87 <applet code="CheckboxGroupAgent.class"
88 name="op" width=60 height=100 align="top">
89 <param name="init_s" value='
90      @update +checkbox(self, "move", "move");
91      @update +checkbox(self, "delete", "delete");
92      @update +checkbox(self, "new text", "newtext");
93      @update +checkbox(self, "new image", "newimage");
94      @addquery checkbox ~LABEL, ~VALUE : checkbox(self, ~LABEL, ~VALUE);
95      @addquery changed ~VALUE : operation(canvas, ~VALUE);
96 '><param name="selected_s"
97          value='@update +operation(canvas, *value);
98 '></applet>
99 <BR>
...
...         ... textfield for typing username, checkboxes for selecting telepointer ...
...         ... list of online users ...
...
129</BODY></HTML>
```

Again, most of the specification consists of queries and updates, with the exception of the creation of the graphic creation dialogs (line 40-85), which is straightforward code which is not listed here.

The canvas is particularly interesting, as the most important things happen here. The canvas displays several things, graphical objects, the selections, and the users' telepointers. The queries for these can be specified in a straightforward manner (lines 9-20). Note that it is also quite easy to implement design decisions concerning visibility here. We may for example decide to make the other users' selections visible, by changing line 17 to:

```
graphic(~OBJ,~X,~Y) selected(~CANV,~OBJ)
canvas(~CANV,_,~COLOUR);
```

The *pointer_moved* script shows how the move and delete operations are implemented. Both could be specified with a single *@updateq* statement, illustrating some of the expressive power of this operation. Note that the condition under which the operation should be done is given by a set of query predicates that specifies that condition (lines 24 and 29). The modal behaviour of the interface is specified here: for each operation, the mode in which it is allowed can be specified in this manner, in this case by the predicate *mode(self,"drawing")*.

Note that the condition that allows the cursor position to change is specified in a slightly odd way: we specify a locally-used value *drawingmode* that holds the value of the desired mode (line 31), which is then matched with *mode* (line 34) to decide if the cursor may move. This somewhat cumbersome detour is required, because we cannot specify a query without variables (i.e. a query that does not specify a result, but only specifies whether something exists or not). This is inherited from an idiosyncrasy of the underlying Prolog engine, and should be fixed in future versions.

If we wished non-modal behaviour, we could have implemented this by passing the desired position of the newly created image to the dialog box. The position could easily be made visible to the user or other users by adding a query to the canvas. Enabling visibility would of course require an entity for the dialog box to be added to the structure specification.

Finally, we will discuss the problems with deleting images, and why we will not allow it in this version. One problem concerns a graphic that refers to an image that is deleted. This requires the graphic to be deleted as well, because otherwise we may fail our requirement that a graphic should have at least text or an image. Such database consistency maintenance could be achieved by specifying a custom reaction that is triggered by the deletion of an image (à la active databases), or by defining a domain agents that alone is allowed to manage graphics and images, and will ensure consistency. Either may be implemented using some extra script code or an extra agent. Since we have not specified this here, we simply disallow image deletion. Summarising, we may state that data structure consistency is not always trivial, and some aspects of it may require careful design, and possibly a more thorough study of database theory is appropriate for creating guidelines for more complex applications.

But there is another reason why deleting images is problematic. When a dialog box for selecting an image is open and an image is deleted, a race condition may occur. The following sequence of events may occur: the user clicks on an image and sends the appropriate creation command to the engine, then the image is deleted, and then the just-selected graphic is created by execution of the creation command. Depending on how exactly the creation command is specified, we may get a graphic that does not have an image, or a relation to a non-existent entity may even be created. For this kind of race condition, where an agent acts on outdated information, the *is_outdated* flag may be used to remedy the situation, and, for example, print an error message or ignore the command.

A second piece of code that is interesting is the *CheckboxGroupAgent* to specify the action (lines 85-98). The checkboxgroup enables its selection to be set from the outside: if the attribute it corresponds to is changed, it updates its selection accordingly (line 95). Note that a somewhat odd flow of data occurs here, which is however unproblematic, and should be quite acceptable in similar situations. When the user presses a checkbox, the following happens: the checkbox is highlighted, indicating the user's selection, then the updated value for *operation* is set by the engine, and then this updated value is communicated as a query result back to the checkboxgroup, which *again* highlights the appropriate checkbox. Interesting concurrent situations may occur, when a user selects items in very quick succession, or other agents update *operation* concur-

rently. But, except from short periods in which the indicated checkbox selection may be outdated or 'echoing' the user's previous selections, eventually it will display the latest value of *operation*. Only in some situations, such as real-time animation such as dragging, will this be problematic because it doesn't look pretty.

8.5.3 Conclusions

To start with, we may indicate that this specification further illustrates some of the power and conciseness of the queries and updates as glue specifications. Several further points were mentioned:

Advantages:

- We may specify modal behaviour of an UI concisely by means of extra terms added to the appropriate queries.
- This application shows the concept of a combination domain and UI data in a single specification to be successful.

Problems:

- We should specify the arguments of multi-argument attributes and events.
- We cannot specify properties concerning changes in VDC. This could be achieved by comparing an old value with a new, updated value.
- It may be useful to have a shorthand for the specification of value ranges in VDC.
- Queries without variables are not allowed. This should be fixed.

Issues:

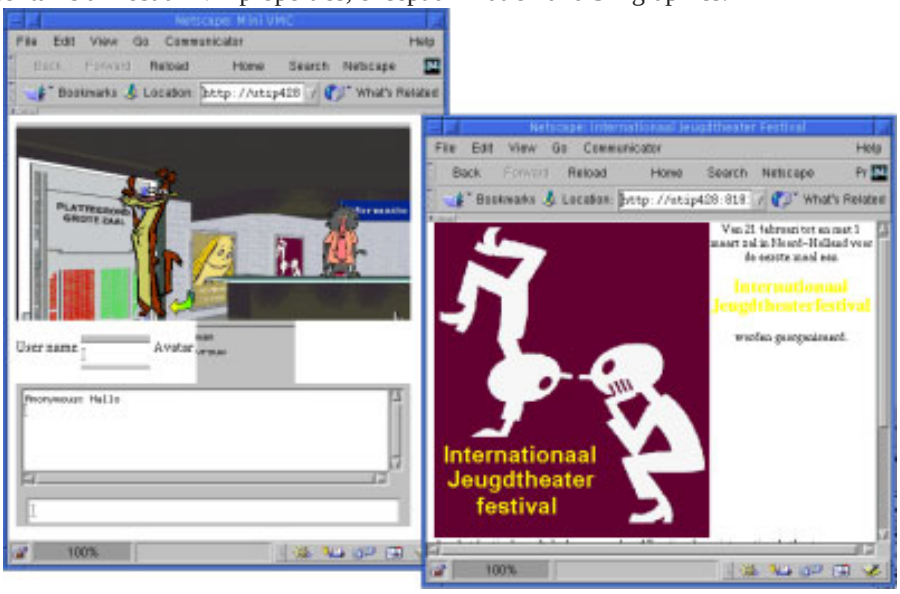
- Managing database consistency for more complex databases is an open problem, for which we may refer to database theory.
- Because our application uses asynchronous communication, the problem of outdated information emerges. We have shown some simple solutions to some of the most basic problems. This includes simply throwing away a script when it is outdated, and the feasibility of changing and tracking a concurrently modified value using plain updates and queries.

8.6 Mini-VMC: VE, Web environment, multimodal dialogue agent

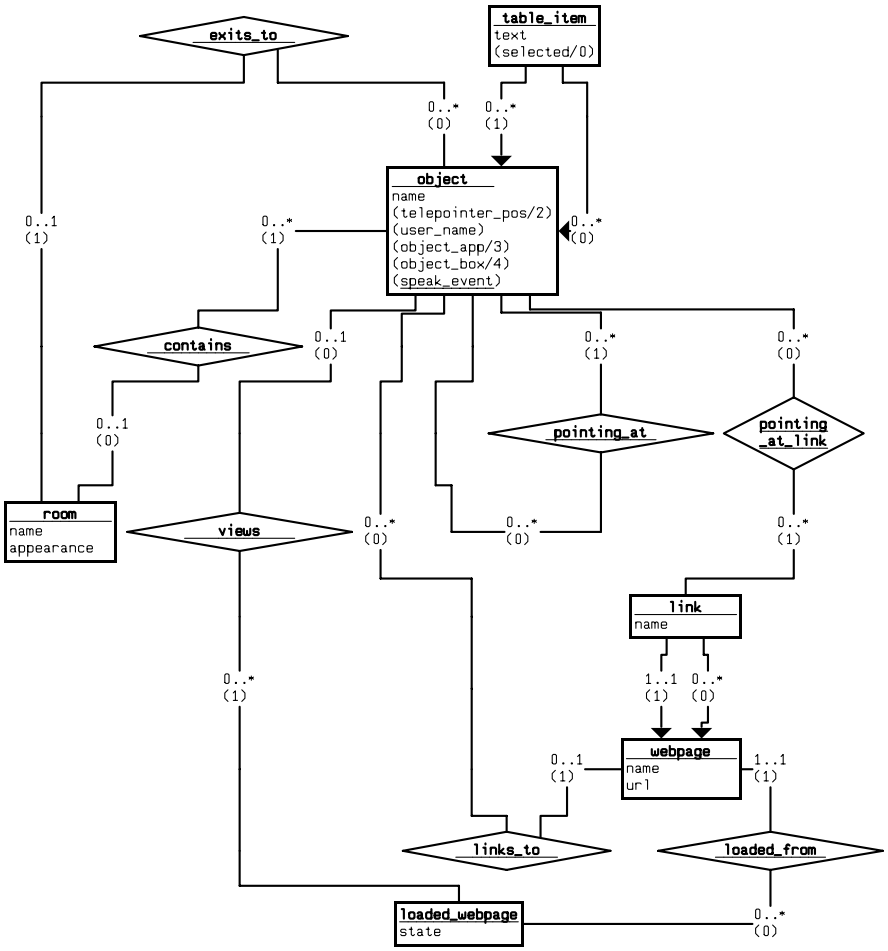
We will continue with a specification of the running example. We will specify a slightly more specific version of it in natural language first.

Mini-VMC is a 2D VE combined with a Web environment consisting of a number of HTML pages. There is a dialogue agent situated in the VE, which is based on the VMC Karin agent. It tracks the user's pointing and Web browsing behaviour and answers queries about theatre performances. It may also show a list of query results, which are displayed in a table. Results may be selected by clicking on them. The VE consists of a set of rooms with objects in them, which is displayed as a background image with the objects displayed as images, placed in front of it. The objects may be users, exits which can be clicked on to move to other rooms, and objects that bring up Web pages when clicked. Karin is also an object. The user may select user name and an avatar, and may move the avatar around by clicking on the room. Karin and the users may chat by means of text.

Mini-VMC is specially meant to model a complex environment in a structured way, so that a multimodal agent can easily be made aware of it. It models both the structure of Web pages being browsed and a VE consisting of a structure of rooms and objects. It contains almost all VE properties, except animation and 3D graphics.



8.6.1 Structure



Mini-VMC is the example with the most complex structure. It describes two information structures: the structure of Web pages being browsed (given by *link*s linking *webpage*s, and the structure of the VE world (given by *room*s, *containing object*s, some of which may *exit_to* other *room*s).

Beside these, there are some structures for describing UI state. First, one should note that avatars and interface agents are *object*s. Agents may be *pointing_at* objects and *pointing_at_link*s. They may have *loaded_webpage*s, *loaded_from* specific *webpage*s, which they may be *view*ing. Finally, and agent may show a *table_item* to another. As this text paraphrases the diagram almost literally, the structure diagram should be quite intuitive to understand.

Note that we have used entities with references to model *table_item* and *link*, rather than use extra relations to link the entities. While this is conceptually not clean, it is quite easy to understand implementation-wise. While an equivalent model may be created

without the entities with references by creating extra relations, using the entities with references simplifies the model. It is for this reason that we allow such constructs, even while they are strictly speaking not E/R modelling constructs.

When we try to separate the data into domain data and UI data, we run into trouble. We may view the VE structure as domain data, as it describes information in our information domain. Hence, *room* and some *object*s are domain data. In a similar vein, we may even state that avatars are domain data. We may wonder, however, whether the Karin agent or the *table_item*s it shows are domain or UI data. One source of trouble here is that Karin encapsulates domain data not in our model, namely, the available theatre performances. When Karin speaks or shows table items, these in fact relate to the unmodelled domain data, and may hence be classified as UI data that refers to domain data. But, just as we may model Karin as UI data, so may avatars, and other VE objects as well, since they represent some kind of interactive objects. This discussion indicates that the separation of domain and UI in VEs may not always be clear. The question is, whether this is really a problem, especially since we integrate both anyway.

The complete VDC specification is given below.

```
1  entity link(webpage:*,webpage:1,name) { }
2  entity webpage(name,url) { }
3  entity loaded_webpage(state) { }
4  relation views(object:0..1,loaded_webpage:*) {}
5  relation loaded_from(loaded_webpage:*,webpage:1) {}
6  relation links_to(object:*,webpage:0..1) {}
7
8  // an agent may point to another object in the scene ...
9  relation pointing_at(object:*,object:*) {}
10 // ... or at a link.
11 relation pointing_at_link(object:*,link:*) {}
12 // objects are usually located in a room
13 relation contains(room:0..1,object:*) {}
14 // some objects may be exits to other rooms
15 relation exits_to(object:*,room:0..1) {}
16 // agents may show information items (typically shown in a table) to users
17 entity table_item(object:*,object:*,text) {
18       // indicates that the user selected the item
19       value selected/0;
20 }
21 // user may only select one item from a table from the same agent
22 set items_selected(~item1,~item2) :
23       table_item(~item1,~agent,~user,_) selected(~item1)
24       table_item(~item2,~agent,~user,_) selected(~item2);
25 assert {items_selected.item1 == items_selected.item2}
26 entity object(name) {
27       value telepointer_pos/2, user_name;
28       // an object may have an appearance or just a bounding box
29       value object_app/3, object_box/4;
30       // avatar objects may speak
31       event speak_event;
```

```
32      set pointing(~object,~name) :
33          pointing_at(self,~object) object(~object,~name);
34      set showing_table(~user,~name) :
35          table_item(_,self,~user,_) object(~user,~name);
36      set exits(~src,~dest) : contains(~src,self) exits_to(self,~dest);
37      set has_app(~image,~x,~y)   : object_app(self,~image,~x,~y);
38      set has_box(~x,~y,~w,~h)    : object_box(self,~x,~y,~w,~h);
39      set has_telepointer(~x,~y) : telepointer_pos(self,~x,~y);
40      set has_url(~topic,~url) : links_to(self,~page) webpage(~page,~topic,~url);
41      // We specify what combinations of values correspond to the different
42      // kinds of objects.  This specification is used as an alternative to a
43      // subclassification.
44      statemachine {
45          Begin -> Visible {} {has_app or has_box}
46          Visible -> IsKarin {} {object.name==”karin”}
47                  -> IsUser  {} {object.name==”user” and user_name}
48                  -> IsExit {} {exits}
49                  -> IsInfo {} {has_url}
50      }
51      // now define the attributes that may or may not be defined in each state
52      assert Begin,Visible { not has_url and not has_telepointer and
53          not speak_event and not exits and not user_name and not pointing_at }
54      assert Visible,IsKarin,IsUser,IsExit,IsInfo  {has_app or has_box}
55      assert IsKarin,IsUser {not has_url and not exits}
56      assert IsExit,IsInfo {not has_telepointer and not speak_event and
57                                  not user_name and not pointing_at}
58      assert IsKarin {not user_name}
59      assert IsUser {       user_name}
60      assert IsExit {not has_url and       exits}
61      assert IsInfo {      has_url and not exits}
62      // only Karin may show tables, to users only
63      assert {showing_table implies
64              (object.name==”karin” and showing_table.name==”user”)}
65      // an exit should exit to a different room
66      assert {not exits.src == exits.dest}
67  }
68  entity room(name,appearance) { }
69  // rooms may not have the same name
70  set rooms(~id1,~id2,~name) : room(~id1,~name,_) room(~id2,~name,_);
71  assert {not rooms.id1 != rooms.id2}
72  // we would like to specify that all rooms are reachable
```

The code should be mostly self-explanatory. Note that we only have one interface agent, called ”karin”, but other agents are modelled analogously. Particularly interesting is the state automaton and corresponding assertions. Here, we specify the attribute lifecycle of the *object* in detail. We are in fact specifying a subclassification, showing which attributes stand for which kinds of objects, and which of the optional attributes may or may not be defined in each kind of object. While subclassifications are a pretty standard extension to E/R models, it was considered unimportant for understanding the

essentials of our technique, and it was not used often in our examples either. It may be added in a future version. Here, we have viewed the suitability of other types of specification to compensate for the lack of subclassification a challenge for our technique. We may have used extra assertion criteria instead of a state automaton here (i.e. we add in front of each assertion: `<criterium_specifying_object_type implies the_other_criteria`). We chose however for this approach because using states makes our intention clearer, and they can be easily referred to in the assertions, leading to a much more concise specification.

We have specified two assertions of the type 'no two objects may have a property X' (lines 21-25 and lines 69-71). We used a standard way to specify this using standard language constructs, but possibly, this type of assertion may be more concisely specified using a special construct, better indicating the intention of the specification.

Another interesting issue arises in the assertion on lines 62-64. What we are specifying is that showing a table implies that the object is of a specific type (*IsKarin*), and the object that it is shown to is of another specific type (*IsUser*). The assertion does however not refer to the object types, but rather, to the 'low-level' properties that are the indicators for them. A cleaner specification may be made by referring to the state of both objects explicitly, which is at present not possible.

There is in particular one thing which we cannot specify (line 72). Reachability of rooms, as we have seen in the logic specifications of chapter 5, requires calculation of the transitive closure. Transitive closure is increasingly often found in database and assertion checking languages, indicating that it is both feasible and useful, and may be added in future versions.

8.6.2 Application

The application consists of a start script (VETkScript), which starts up the Karin agent and creates the world structure, the Karin agent (VCL), the VE view (HTML), which is the main user interface through which the user interacts, and some HTML pages containing information, and the table that shows Karin's results. We will show a part of the VE view here, as well as the Karin start script. The other specifications are straightforward.

The VE view consists of a canvas, and various standard GUI elements to set attributes, and view and type chat texts. The specified scripts consist entirely of addquery's and updates. We will only specify the canvas here.

```
1  <HTML><HEAD> <TITLE>Mini VMC</TITLE> </HEAD><BODY>
2  <!– The canvas displays the objects in the current room –>
3  <applet code="ObjectCanvasAgent.class"
4  width=400 height=200 align="middle">
5  <!– show, respectively, telepointers, objects, rooms, and bounding boxes –>
6  <param name="init_s" value='
7       @addquery graphical_object ~USER, "ptr", "telepointer.gif", ~X, ~Y :
8            contains(~ROOM, avatar) contains(~ROOM, ~USER)
9            telepointer_pos(~USER, ~X, ~Y);
```

```
10      @addquery graphical_object ~OBJ, "object", ~IMG, ~X, ~Y :
11          contains(~ROOM, avatar) contains(~ROOM, ~OBJ)
12          object_app(~OBJ, ~IMG, ~X, ~Y);
13      @addquery graphical_object ~ROOM, "Room", ~IMG, 0, 0 :
14          contains(~ROOM, avatar)
15          room(~ROOM, _, ~IMG);
16      @addquery bounding_box ~OBJ, "box", ~X, ~Y, ~W, ~H :
17          contains(~ROOM, avatar) contains(~ROOM, ~OBJ)
18          object_box(~OBJ, ~X, ~Y, ~W, ~H);
19  '><param name="pointer_moved_s" value='
20      /* make telepointer follow mouse pointer */
21      @update +telepointer_pos(avatar, *xpos, *ypos);
22  '><param name="pointing_at_s" value='
23      /* update pointing_at when the user moves the mouse over an object */
24      @update state pointing_at(avatar, source);
25  '>
26  <!– What happens when user selects an object –>
27  <param name="selected_s" value='
28      if (state==1) : {{
29          /* private use only */
30          @update state _selected(self, type);
31          /* follow an exit when clicked */
32          @updateq +contains(*~ROOM, avatar) : _selected(self, "left")
33              pointing_at(avatar, ~OBJ) exits_to(~OBJ, ~ROOM);
34          /* bring up page when object links to page */
35          @updateq .show_document(avatar, ~URL) :
36              pointing_at(avatar, ~OBJ) links_to(~OBJ, ~WEBPAGE)
37              webpage(~WEBPAGE, _, ~URL);
38          /* move avatar using right mouse button */
39          @updateq +object_app(avatar,~APP, ~X, ~Y) : _selected(self,"right")
40              telepointer_pos(avatar,~X,_) object_app(avatar,~APP,_,~Y);
41      }}
42  '></applet>
...
...         ... type user name, select avatar, chat output, ...
...         ... type messages, handle show_document requests, ...
...         ... track user's web pages ...
...
98  </BODY></HTML>
```

There are few surprises here, as the objects visible on the canvas can be specified in a straightforward manner (lines 5-18), and so can the operations that should be done when the user clicks on something (lines 26-41).

The start script is given below.

```
1   // Web pages
2   @new ijtf;
3   @update +webpage(ijtf,"ijtf","links/posters/ijtf.html");
4   @new assep;
```

```
5  @update +webpage(assep,"assep","links/posters/assep.html");
6  @new link1;
7  @update +link(link1,ijtf,assep,"assep");
8  // Rooms
9  @new front;
10 @update +room(front, "front", "vmc_front.gif");
11 @new foyer;
12 @update +room(foyer, "foyer", "vmc_foyer_l.gif");
...
...        ... create some clickable objects and exits ...
...
33 // The Karin agent
34 @new karin KarinAgent init_s = <<
35     // track the objects and links the user is pointing at.
36     // Object with URL is information object.
37     @addquery user_pointing ~USER, ~TOPIC :
38         pointing_at(~USER, ~OBJ) object(~OBJ, ~TOPIC) links_to(~OBJ,_);
39     @addquery user_pointing ~USER, ~TOPIC :
40         pointing_at_link(~USER,~LINK) link(~LINK, _,_, ~TOPIC);
41     // track the webpages the user is viewing
42     @addquery user_viewing ~USER, ~TOPIC :
43         views(~USER, ~LPAGE) loaded_webpage(~LPAGE, _)
44         loaded_from(~LPAGE,~PAGE) webpage(~PAGE,~TOPIC,_);
45     // track the users' utterances
46     @addquery user_utterance ~USER, ~UTT :
47         contains(~ROOM, self) contains(~ROOM, ~USER)
48         speak_event(~USER, ~UTT);
49     @addquery item_selected ~USER, ~TEXT :
50         table_item(~ITEM,self,~USER,~TEXT) selected(~ITEM);
51 >> utter_s = <<
52     @update .speak_event(self, utterance);
53 >> new_table_s = <<
54     // initialise a new table: clear its items and show a webpage which
55     // displays any items added later
56     @update -table_item(_,self, user, _);
57     @updateq .show_document(~PAGE, "table_view.html") : views(user, ~PAGE);
58 >> table_item_s = <<
59     // add item to existing table
60     @new itemid;
61     @update +table_item(itemid, self,user, item);
62 >>;
63 @update +object(karin, "karin");
64 @update +contains(foyer, karin);
65 @update +object_app(karin, "karin_avatar.gif", 306, 74);
```

The creation of web pages, rooms, and objects is done by straightforward updates, such as found in lines 1-12. The Karin agent is initialised in lines 33-62, which amounts to the initialisation of its views and scripts. The views are comprised of: viewing the user activities *user_pointing* (the VE objects and hyperlinks a user points to), *user_viewing*

(the Web pages a user is viewing), *user_utterance* (the user's NL utterances), and *item_selected* (the table items that a user selects). Karin's output scripts, defined in line 51-62, define how Karin's utterances and tables are processed. When a table is shown, first, *new_table_s* is called, and then, *table_item_s* is called once for each item in the new table.

8.6.3 Conclusions

We may again state that the specification was successful, specifying what we meant to specify in a concise manner. We mentioned some further points:

Advantages:

- In terms of structure modelling, this applications showed best that our structure modelling technique results in comprehensive specifications.

Problems:

- We may require a more tailored specification for requirements of the form: 'no two items may have property X'.
- In this example, a standard E/R subclassification mechanism would have been useful.
- It would have been useful to be able to refer to the automaton states of any objects from within assertions.
- This example required a transitive closure operation.
- Are entities with references a proper structural construct? We chose this somewhat unusual construct to specify a relation with extra attributes. This is because we choose relations not to have a separate identity value, as this is usually not needed.

Issues:

- In this example, we tried to show that it is questionable whether the traditional separation between domain and UI data is meaningful.

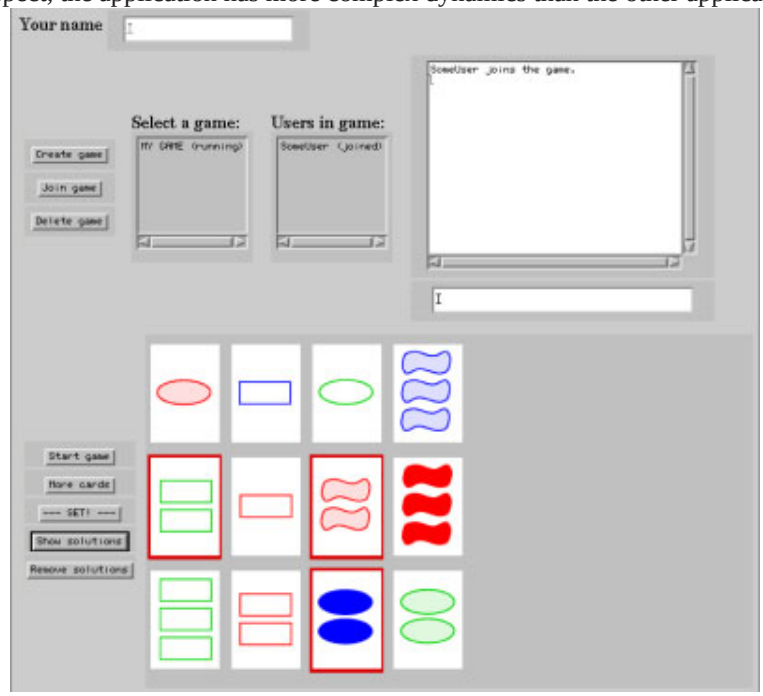
8.7 Webset: a multi-user card game

Webset, a multi-user application based on the card game *Set*, is the last example we will discuss here. It was also featured in a paper (van Schooten, 2002).

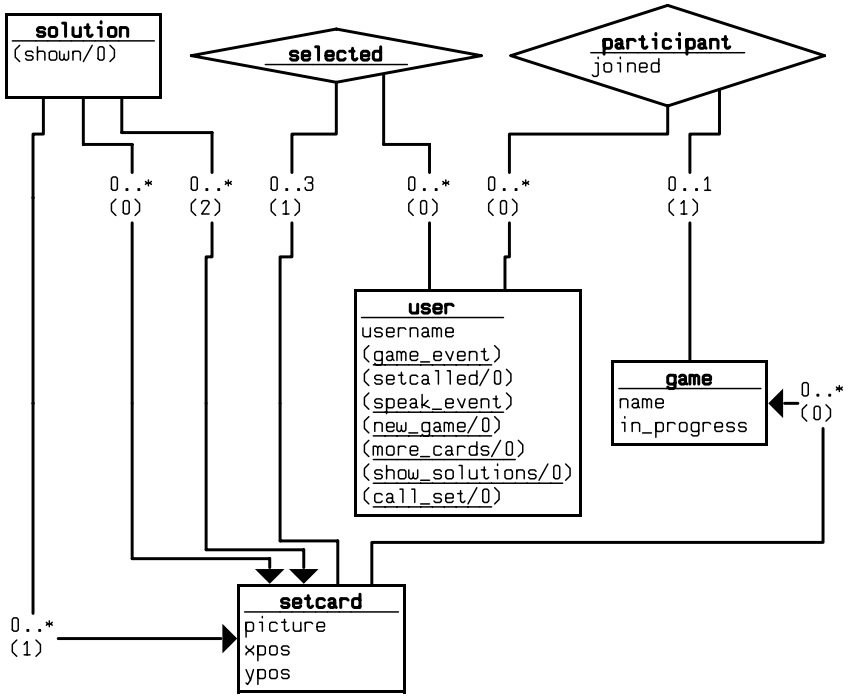
Users are able to log in via the Web, after which they are presented with an entrance screen. Here, they can fill in their name, create and delete games, or join games that

have not yet begun. When a game is joined, the user can talk to the other users that have joined the game. Once a game is started, the cards being played with are displayed. The game proceeds as follows: 12 cards are laid out from the deck on the table. Now, the players have to search for combinations of three cards that satisfy certain criteria; such a combination is called a ‘set’. The first one to see a set calls ‘set!’ and then has a couple of seconds to point out the three cards. These are removed and replaced with cards from the deck. The game continues until the deck is exhausted. The player with most sets wins. When no player can find any sets, extra cards may be laid out. The game should also have the option to point out sets, pausing the game, and enabling all players to see the sets on the table, after which the game may be continued with fresh cards.

Webset is an application with a slightly different nature than the other three, and we include it here to illustrate some further issues. In particular, it includes a set of game rules, and some turn-taking policies, and uses a special agent that manages the game. Turn-taking policies may be seen as a generalisation of modal interface behaviour. In this respect, the application has more complex dynamics than the other applications.



8.7.1 Structure



The *user* and *game* entities should be obvious, but some of the *user*'s attributes and events require explanation. *game_event* indicates that something happens in the game that concerns that specific user, such as his/her success or failure of getting a set. *setcalled* indicates that the user is now allowed to select cards to form a set. The other events indicate basic user interface events: typing a message, starting a new game, requesting more cards, requesting the game to show solutions, and calling 'Set'. Note that most of these events are just messages, signalling requests to the game.

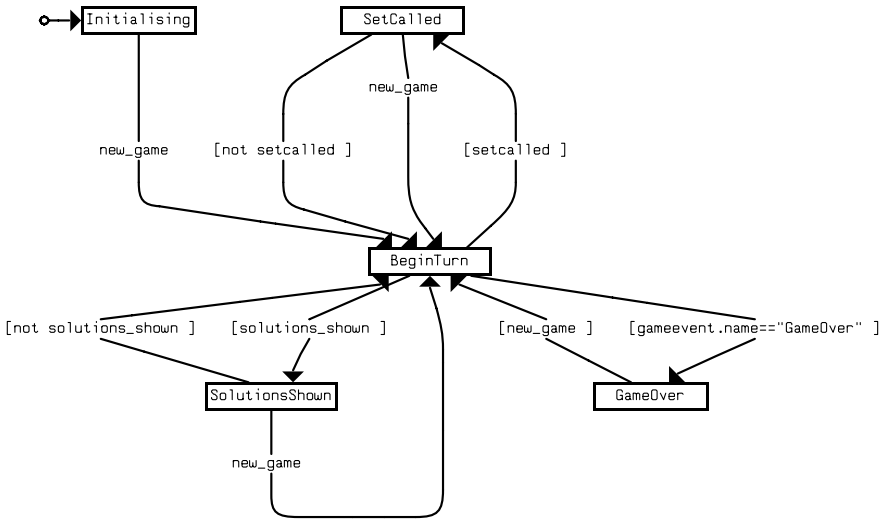
Participation in games is modelled by the *participant* relation. A user may participate in at most one game, and any number of users may participate in a game. A game has a number of cards on the table, indicated by the *setcard* entity. Finally, when the system is displaying solutions, for each solution a triple of setcards is related through the *solution* entity. We have again modelled this as an entity with references, because it contains an optional attribute.

In a game too, we may separate the domain and the UI part of the structure, when we consider the game's state domain data. *Setcard*, *game*, and *solution* may be considered domain data, while *user* and its relations are UI data. Exceptions to this is the attribute *shown*, which may be considered UI data, and *game_event*, which may be considered domain data, because it signals what is going on in the game. While the domain/UI separation is clearer here than in Mini-VMC, we also see that single attributes in an entity of one type may be of the other type.

```
1  // Entity user specifies a player that may participate in games
2  entity user(username) {
3       // Game information related to this user, signalled by the game
4       event game_event;
5       // Indicates that user may select cards after calling set.
6       value setcalled/0;
7       // The user may speak to other users
8       event speak_event;
9       // The requests that the user emits to the game
10      event new_game/0,more_cards/0,show_solutions/0,call_set/0;
11      set player(~game,~joined) : participant(self,~game,~joined);
12      // The lifecycle of a user.  A user may not participate in any game
13      // (Free), may watch a game (Watching), or join a game (Playing).
14      statemachine {
15           Free -> Watching     {} { player.joined==0 }
16           Watching -> Playing     {} { player.joined==1 }
17                    -> Free         {} { not player }
18           Playing -> Free       {} { not player }
19                    -> Watching {} { player.joined==0 }
20      }
21      // When the user is Free or Watching, it may not receive game_events and
22      // permission to select sets.
23      assert Free,Watching { not game_event and not setcalled }
24      // value range assertion
25      assert {game_event.val==”AlreadyCalled” or game_event.val==”NoCall” or
26              game_event.val==”GotSet” or game_event.val==”BadSet” or
27              game_event.val==”CallExpired” or game_event.val==”GameOver” or
28              game_event.val==”InternalError” or not game_event}
29 }
30 entity game(name,in_progress) {
31      set new_game(~player) : new_game(~player) participant(~player,self,1);
32      set setcalled(~player,~joined) :
33           participant(~player,self,~joined) setcalled(~player);
34      set solutions_shown(~cd1,~cd2,~cd3) : solution(~cd1,~cd2,~cd3)
35           setcard(~cd1,self,_,_,_)
36           setcard(~cd2,self,_,_,_)
37           setcard(~cd3,self,_,_,_);
38      set userselected(~player,~card) :
39           selected(~player,~card) participant(~player,self,_);
40      set gameevent(~player,~name) :
41           participant(~player,self,_) game_event(~player,~name);
42      // Lifecycle of game.  A game may be just created (Initialising), at the
43      // beginning of a turn (BeginTurn).  Then a ‘set’ call may be granted
44      // (SetCalled) or a request to show solutions may be done (SolutionsShown).
45      statemachine {
46           Initialising -> BeginTurn { new_game }
47           BeginTurn -> SetCalled         {} { setcalled }
48                     -> SolutionsShown {} { solutions_shown }
49                     -> GameOver {} { gameevent.name==”GameOver” }
50           SetCalled -> BeginTurn      {} { not setcalled }
```

```
51                      -> BeginTurn      { new_game }
52          SolutionsShown -> BeginTurn {} { not solutions_shown }
53                        -> BeginTurn { new_game }
54          GameOver -> BeginTurn {} {new_game}
55      }
56      assert Initialising,BeginTurn { not userselected and not solutions_shown }
57      assert SolutionsShown { not userselected and not setcalled }
58      assert GameOver{not userselected and not solutions_shown and not setcalled}
59      // only one player which has joined may be granted the set call
60      assert { size(setcalled) <= 1 and not exists(setcalled.joined==0)}
61      // only the player who is granted a set call may select cards
62      assert { setcalled implies
63          (userselected and not exists(userselected.player!=setcalled.player)) }
64 }
65 entity setcard(game:*,picture,xpos,ypos) {}
66 relation participant(user:*,game:0..1,joined) {}
67 relation selected(user:*,setcard:0..3) {}
68 entity solution(setcard:*,setcard:*,setcard:*) {
69      // indicates that solution is shown on screen
70      value shown/0;
71 }
72 // Solutions must be of different cards which must be in the same game.
73 set solutions(~game1,~cd1,~game2,~cd2,~game3,~cd3) : solution(_,~cd1,~cd2,~cd3)
74      setcard(~cd1,~game1,_,_,_)
75      setcard(~cd2,~game2,_,_,_)
76      setcard(~cd3,~game3,_,_,_);
77 assert{ not exists(
78      solutions.game1 != solutions.game2 or solutions.cd1 == solutions.cd2 or
79      solutions.game1 != solutions.game3 or solutions.cd1 == solutions.cd3 or
80      solutions.game2 != solutions.game3 or solutions.cd2 == solutions.cd3) }
81 // we would like to specify much more with respect to the game rules, such as
82 // that a player may only remove cards that are actually sets, that the deck
83 // of 81 cards is laid out until the game ends, that solutions shown are
84 // indeed sets, etc.  We leave specifications related to the game cards to the
85 // game agent.
86 // Neither did we specify what game_events may happen when, but this is again
87 // a matter decided by the game agent alone.
```

This specification defines a couple of interesting state automata. We specify the participation modes for *user*, and the events and values that may and may not be defined for each mode (lines 12-20). The other automaton, in *game*, specifies the turn-taking and floor control policies of the game, and is large enough to warrant showing the generated diagram as well:



When the game has been initialised, there are four states. *SetCalled* indicating that a user has called ‘set’ and we are waiting for the user to select cards or the call to expire. *SolutionsShown* indicates that the game is frozen while the computer shows the possible solutions. Finally there’s *GameOver*, which is entered when the ”GameOver” event is signalled, and can only be exited when a new game is started.

There are several more assertions that specify proper game behaviour. There’s another range assertion (lines 24-28), an assertion specifying desirable properties of the solutions’ relations (lines 72-80), and assertions concerning the set call (lines 59-63).

At the end (lines 81-85) we state that we did not manage to assert various game properties. These mostly concern properties of the cards, which would require lengthy specifications. It is acceptable not to specify it in the global data specification, as it is managed by the game agent alone. The same goes for the game events. While we have chosen to specify as much as is feasible (that is, without straying too far from our goal of specifying user interface structure with verbose specifications), we could leave out anything that is entirely under the control of the game agent, as long as we replace the assertions with appropriate access control policies (which, in our current specification, would have to be specified by convention rather than formally).

This brings up another question: who ensures what properties? Does the game agent manage and check everything, with the users just sending dumb requests through pressing buttons without checking if they are meaningful? Or is the user interface ‘smart’, and does it disallow illegal actions, or better, indicate illegal actions pre-emptively, for example by disabling buttons which are not appropriate in a specific state? This issue implies a certain trade-off of SE issues with usability issues here, which has also been identified by others (Evers, 1999). Since we specified much of the game policies in our global specifications, we may more easily choose to distribute responsibilities across the different agents. Failure to meet responsibilities will be trapped by the assertion checks to some extent. While the SE / usability trade-off is still present, it is reduced by postponing the decisions to after the structure specification phase.

8.7.2 Application

The application consists of the user interface (HTML), a startup script that initialises the game agent (VETkScript, called when a user creates a new game), and the game agent (VCL). We will only show a few bits of specification that point out a couple of interesting issues.

The agent acts on messages it receives from users, in the form of events. The handlers for these events use the events' parameters just as regular procedure call parameters. They react on just the addition (or alternatively, the deletion) of the set element standing for the event:

```
1   imports {
2         java.util.*,
3         java.awt.*
4   }
5   agent SetGameAgent(
6   Integer xtile=new Integer(80), Integer ytile=new Integer(112),
7   String del_card_s, /* del_card(String card) */
8   String add_card_s, /* add_card(String card,Integer xpos,Integer ypos) */
9   String solution_s, /* solution(String card1,String card2,String card3) */
10  String user_scoring_s, /* user_scoring(Object user,String status) */
11  String set_called_s,     /* set_called(Object user) */
12  String game_over_s       /* game_over()*/
13  ) {
14        handle new_game() { if (is_addition) {
...
...         ... body ...
...
22        } }
23        handle show_solutions() { if (is_addition)
...
...         ... body ...
...
32        } }
...
...         ... the rest of the agent body ...
...
216}
```

We have already suggested a special notation for event handlers in chapter 7. Though it is clear here that the lack of these does not provide much specification overhead, this may be added in a future version. Note also the use of comments to specify script parameters (lines 7-12), which is another problem indicated in chapter 7, that should be solved using a special parameter type for scripts.

Another issue is that we found the need to refer to the state automaton states in the glue specifications. For example, since the user may only participate in game activity when in its *Playing* status, queries meant to specify Playing users only may be most naturally specified by stating that the *participant* should be in that particular state. In

several places, we had to implicitly refer to automaton states by specifying a condition implicit to those states. For example:

```
1  // Creates a new game.
...
...        ... textfield creates this agent when game name is entered ...
...
13         // the game agent, a domain agent (i.e. a MVC Model component)
14         @new setgame SetGameAgent width=0, height=0,
15         init_s = <<
16               // listen to requests done by participants
17               @addquery new_game ~USER :
18                     new_game(~USER) participant(~USER,self,1);
19               @addquery user_calls_set ~USER :
20                     call_set(~USER) participant(~USER,self,1);
21               @addquery show_solutions ~USER :
22                     show_solutions(~USER) participant(~USER,self,1);
23               @addquery end_show_solutions ~USER :
24                     end_show_solutions(~USER) participant(~USER,self,1);
25               @addquery user_requests_new_cards ~USER :
26                     more_cards(~USER) participant(~USER,self,1);
...
...        ... the rest of the agent, the rest of the specification ...
...
92 @wadd(root,win);
```

We specify here that participants that signal messages to the game engine must be *Playing* participants. The condition `participant(~USER,self,2)` refers to the *Playing* state. This indicates that the state automaton specifications are not just meaningful as correctness constraints, but also as executable code.

8.7.3 Conclusions

This application could be specified sufficiently well, though it did pose some interesting challenges.

Advantages:

- This is the application with the most complex layout, including grid-like layout, which is easily specified using HTML tables.

Problems:

- As indicated before, we showed that regular message handlers should be defined as a special kind of handler.
- The availability of automata states from scripts (i.e. having them as tuples in the database) may be useful.

- As indicated before, we identified the need to define script parameters.

Issues:

- In this application, there is a complex agent that manages the game rules. Do we need to specify constraints globally, when they are managed by only one agent? A related issue is: who checks the satisfaction of what data constraints? We have indicated that this is a matter of design decision. Possibly, a combination of global specifications with access policies would produce a comprehensively complete correctness specification.

8.8 Some general conclusions

We have assessed the technique using examples, and illustrated advantages, problems, and issues that surfaced with the specification of each example. The issues discussed here mainly concern content suitability. We will now discuss some general findings concerning process suitability, i.e. the things encountered while the above specifications were under development. While it was not our intention to do a systematic evaluation of the development process, still, we can say some useful things in this area, using the anecdotical evidence obtained during the development of these examples.

All of the examples were first written in an early version of VETk, that did not include the VDC language, but just the executable languages. While the suggested development methodology was in fact derived from development practice, the VDC and state automaton specifications were not part of the development process of the early version. Instead, an informally specified ERD was used. Throughout the development process, the ERD was often used as reference material, and is obviously one of the most important documents in our technique.

The later addition of VDC does make it easier to assess the difference that this formal data specification makes. The formalisation of the ERD part in particular provided insight in all of the applications, and showed that the early informal ERDs were incomplete and imprecise. The full formal modelling of the ERDs prompted several design changes: for example, to remove redundant relations, fill in implicit references, and tidy some misplaced data.

Specification of the state machines, something that was not done at all in the early versions, also provided insight and prompted some redesigns. Especially the automata in the Webset specifications prompted improvement of the modelling of the user and game lifecycle, clarifying possible transitions and possible actions in each state. In fact, the Webset example was one of the incentives for producing state automaton specifications.

The close coupling of the specification with the executable system provided some extra insight, showing that such a coupling is indeed useful. Assertion violation detection helped in adjusting the specification and the implementation to each other.

Concurrency problems surfaced little during development, and low-level concurrency problems (such as inconsistency of data resulting from concurrent access) were non-

existent. Yet, concurrency was sometimes an issue as asynchrony makes some solutions more difficult. As indicated in the examples, we did manage to solve some basic problems using basic solutions.

Chapter 9

Conclusions and future directions

9.1 Introduction

In this chapter, we will look again at our original research plan, and summarise and discuss the achievements and the things not achieved in the light of this plan. This discussion will progress into increasing levels of detail, as we get closer to our research focus of specification languages. We will describe our detailed findings concerning the specification technique along with their most obvious solutions. We will conclude this chapter with some specific future directions.

9.2 Achievements and unaddressed problems

The overall plan that we have set up consisted of identifying development tools related to the development of those interactive computer systems that have the properties we identified: graphical interaction, multimodality, interface agents, and multi-user. Additionally, we gave a list of interesting state-of-the-art VE systems. We identified five major classes of development tools: methodology, guidelines, specification, analysis, and development environments. We have given an overview of existing best practices of these five. With help of this relatively complete overview of the field, we were able to define an outline for future research. In this chapter, we will summarise the possible future directions.

We have evaluated some specification languages and techniques, and created our own specification technique based on these techniques. The new specification technique is designed to be especially useful for: GUIs, UIs with complex structure, limited multimodal UIs, limited interface agents, and multi-user UIs. Multimodality is limited in the sense that no multimodal interpretation facilities are provided, nor is input from

special devices, such as speech input. Interface agents are limited in the sense that no special behaviour specification facilities, such as a specialised language, is provided. So, what is left to do is: cooperative behaviour specification of assistant agents, NL processing and dialogue, speech and multimodal input processing, and 3D graphics and animation. We have already started work on the latter, see section 9.3.

While the specification technique has been shown to be successful, it is still in its early design stages. A real evaluation of the specification technique (such as evaluation with a group of developers) should be done. First though, there are various things that can be fixed without such an evaluation. In fact, various non-essential imperfections in the current version of the technique could lead to problems when trying to evaluate it.

We have provided a relatively complete implementation of a specification toolkit. In terms of the other four classes of development tools, though, we can identify various things that are left to be done.

- **Methodology.** We gave a preliminary methodology, but this was just an illustration, and a real methodology should be worked out in more detail. In order to do this, further research and experience with real-life methodology is required.

- **Guidelines.** We gave a very short introduction to development guidelines, mostly basing ourselves on established practices, but a more thorough coverage could be obtained by extensive literature research. For instance, there is a lot to be found in the literature, especially on GUIs. While we chose to work within an existing GUI framework that is not exactly state of the art, this framework has various usability issues that can further be addressed, such as focus control, menus, scrolling behaviour, etc.

  For some of the VE properties, novel guidelines may need to be developed: we noted that especially guidelines for creating 3D graphical environments are lacking in the literature.

- **Analysis.**

  Descriptive analysis. We did not cover the subject of descriptive analysis, as we decided to work with fixed requirement specifications defined beforehand. As we have indicated in chapter 3, descriptive analysis languages could have some overlap with design specification languages. Existing descriptive analysis languages, such as task modelling languages, could be incorporated in the specification technique. This is an interesting direction to research. However, a proper evaluation of descriptive analysis would require actual users and an actual setting.

  Evaluation. We did not cover any evaluation techniques beside using information obtained through execution and assertion checking. We did not cover usability evaluation, which again requires actual users and an actual setting.

- **Development environments.** We did not much emphasise this type of tool, and further literature research is required here. The only software aid made available in the spirit of development environments are syntax colouring definitions.

9.2.1 Summary of findings of the current specification technique

The specification technique was created after the need was identified to provide an integrated technique for addressing all identified VE aspects. It is based on assessment and selection of the various useful specification languages that were examined in chapter 6.

We claimed that focussing on the data structure of the UI provided an interesting starting point. We also chose to use a component-based architecture and a glue language approach, as this follows recent developments in interactive systems development within SE. We chose an integration of executable specifications with assertion checking specifications as our main specification technique, because we argued that the high level of well-definedness and computational tractability provided by such an approach is essential.

We indicated the feasibility of this idea by creating a specification technique, which was illustrated and discussed by means of example specifications. We summarise our detailed findings in the following paragraphs.

We successfully achieved a number of things. This is a summary from chapter 8.

- Concise data constraint specifications. It is easy to produce ERDs, and various constraints could be specified concisely. The separation of set declarations from assertions (with respect to OCL) was a good idea, and led to improved conciseness of logic assertions and state transitions.
- Comprehensive structure modelling. The UI data structure was found to be useful reference material. We indicated that the idea of combination of domain with UI data is successful, especially since the separation between these two is not always meaningful in the first place.
- Concise glue specifications. Most of the glue could be written as just queries and updates, and no control flow. Modal behaviour and turn-taking and floor control could be specified by just adding extra terms in queries and updates.
- It is easy to add environment-aware interface agents. Agents that observe the user(s) can be added without modifying the rest of the program.
- It is easy to specify distributed multi-user systems. The technique avoids the most basic distributed systems problems, and it is easy to enable visibility of the activities of other users (such as telepointers, etc.).
- HTML provides good layout specification within the textual layout specification paradigm, when compared with more traditional ‘layout manager’ approaches. It provides standard layout constructs such as text-flow-like layout and grid layout using standard HTML tags. Additionally it provides text labels and easy integration with regular Web pages.
- Concise component specification. While the component language offers little in the way of novel features as compared to regular programming languages, agents could be specified in a concise manner.

We identified a number of concrete problems. We present a summary from chapter 7 and 8. We classified the problems by aspect.

General:

- Agents should have an exit_s script.
- There should be some kind of agent-delete command.
- The old value of a set is not available but may be useful in both VDC and VCL.

VCL:

- Generation of the interface (i.e. the set of functions and their parameters) from VCL would be useful.
- A special construct indicating zero-or-one-element set would be useful.
- One should be able to specify regular message handlers separate from database view handlers.
- Enable specification of a script's parameters.

VetkScript:

- Forbid the use of tuples that are not defined in the VDC specification.
- Enable more powerful queries, including a *not* operator to exclude elements from a set.
- Scripts should explicitly specify their input variables.
- The database should provide read/write access policies.
- Queries without variables should be supported.
- The VDC automata states should be made available in the database.
- The query expressions should support transitive closure,

HTML:

- Applet tags are a bit verbose and may be made more concise if we generate the HTML from other specifications.
- Applets cannot define their own window size.

VDC:

- A way to specify pre- and postconditions is needed.
- Values and events with multiple arguments should specify names for their arguments,
- An argument's range should be specifiable as a range of values, such as a set of strings.

- There should be a better means to specify ‘no two items may have some property X’
- A class hierarchy or specialisation construct may be added to the E/R model.
- Automaton states of one entity should be referrable from any other entity.

9.3 Future directions

9.3.1 Suggested redesigns of the languages

Most of the various problems identified with the specification technique suggest straightforward solutions. They will however require various design decisions, and some of them may have to be re-evaluated. The implementation is likely to take some time.

One particularly major redesign involves the integration of VETkScript with VDC. Since VDC is an extension to VETkScript, it seems natural to move VDC constructs that are useful in VETkScript towards VETkScript. In fact, this could go as far as to integrate them into one language, which could then be used for both purposes of the original languages. For example, the set notation looks much like the query notation, and both could be specified using the same notation. Furthermore, the assert statements could be used in VETkScript to specify pre- and postconditions in scripts. Finally, the state automaton states could move to VETkScript, and also communicate their states to the database, so that some state automata become part of the executable specifications where desired.

9.3.2 Open issues uncovered by the example applications

In chapter 8, we identified some open issues, not directly indicating problems with the specifications, but rather, directions for future research:

- We noted the general issue of database consistency. In more complex cases, this requires very careful modelling. The consistency issues that may emerge in more complex applications is a topic of future research.
- We questioned whether the separation of UI data and domain data is always meaningful. While this separation is central to the traditional Seeheim-like models, we found that in some applications, especially in VEs, there is data that we cannot properly classify as either UI or domain data. This questions the validity of these models, which brings up the issue of whether there are particular applications in which separation architectures are not very meaningful.
- How do we specify shared data and access policies so that we obtain a tractable specification? In contrast with more regular models, which are based on greatly limiting access by means of access control policies (i.e. encapsulation and data

hiding), we argued for a global shared data structure. This means that this shared data needs to be well specified, which is done by the VDC language.

In the example applications, we attempted a full coverage of correctness constraints in the global VDC specification, enabling correctness verification of any ensemble of agents working on the data. In some cases though, it may be more practical not to specify correctness constraints globally, but use the more traditional access limiting method of ensuring correctness. For example, in the Webset application, we decided to not to specify the correctness of the data managed by the game agent only, because this would result in difficult and lengthy VDC specifications. More generally, a combination of access control policies with global constraints may be used. The nature of this trade-off as well as more formal underpinnings concerning the tractability and correctness of such a method is a topic of future research.

- How do we use the asynchronous model effectively? The asynchronous message passing model that we used is both reasonably efficient and easy to realise, but it causes the agents to have possibly outdated views on the database. We gave some simple solutions for some practical problems that emerge, but cases can be imagined in which asynchrony is more problematic. While asynchronous message passing models have been well studied theoretically, there are relatively few applications that use an asynchronous model. Most such applications are limited to 1-to-1 communication, such as asynchronous messages between a server and client. There appears to be less experience with asynchronous communication between multiple parties.

  For example, how do we manage a display that both manipulates and tracks a changing database value? This issue occurred on several occasions, and we may assume this is a recurring problem in graphical interaction. Just showing the latest value works reasonably well, but may not look pretty when message delays are notably large. It is possible that, in practice, we may cover most problematic situations by just a limited number of generalised problem patterns. Possibly, we may arrive at a set of design patterns for such issues.

9.3.3 Further future directions

We are working on a 3D and animation modelling language. A first prototype has been built already which shows how 3D graphics may be easily integrated in a VETk application. The current prototype model includes simple collision handling and mouse manipulation functions. 3D graphics are displayed through a special 3D canvas agent, which incorporates the 3D and animation engine. A canvas agent loads and unloads 3D objects according to a database view which indicates the Java classes and ID tags of the 3D objects to be loaded. Once loaded, 3D objects may specify database queries and execute VETk scripts to interact with the shared database. 3D objects are currently written in Java, and the 3D model is implemented as a software library rather than a programming language. However, as was the case with the other specification languages featured in VETk, it is likely that the replacement of the software library with an

appropriate specification language may greatly improve conciseness, and is a topic of future research. The 3D model uses a VRML-like object structure: a 3D scene consists of a hierarchy of graphical objects, each of which has a position and orientation relative to their parent objects. Each object may define a set of parameters, which determine its appearance, position, behaviour, etc. Each object may incorporate instances of other objects, and may define the values of their parameters by means of Amulet-like formulas, which determine how the values are calculated from other variables anywhere in the object tree. When a variable is changed, the 3D engine determines which formulas have to be recalculated. Formulas may define mutual dependence between variables, in which case a cycle-breaking scheme is used to determine which variables have to be updated.

One problem we did not really address but which is worth investigation is the support for various alternative parallel/distributed communication schemes. For example, the 'parallel' (Reactive-C) scheme could be useful, as well as a synchronous form of publish-subscribe. The latter is a non-trivial problem, as some even stated that publish-subscribe is inherently asynchronous (Eugster et al., 2000), which is not strictly true under all conditions. However, a synchronous publish-subscribe system becomes problematic when the information dependencies between the agents contain cycles. While a straightforward cycle-detection scheme seems the most obvious solution, there are actually different options which may be preferable.

While we have examined some languages for specifying intelligent behaviour, we do not provide such a language. Such a language should be part of the technique eventually. An integration of Prolog and production rule systems in the technique is an interesting direction. These languages may also be useful in analysis specification and user simulation techniques.

The integration of the glue with the agents could be made more language-independent, by having VCL take the role of an interface language. VCL would only specify the coupling of the agent with the glue only, with the agent's behaviour being specified in an arbitrary language.

Part IV

Appendices

Appendix A

The VETK technique

A.1 Introduction

In this appendix, we will give a short overview of all the details of the VETk specification technique, usable as a reference manual.

We will use BNF notation for some of the language definitions. Names enclosed by the $<$ and $>$ symbols indicate subclauses. Square brackets indicate optional expressions. The pipe symbol '|' indicates choice. SequenceOf[...] indicates a repetition of its argument of zero or more times, ListOf[...] indicates a likewise comma-separated repetition. Literal strings are enclosed between quotes.

We will start with the execution model for agent initialisation and exit, and update notification, and then proceed with the language definitions.

A.1.1 Agent initialisation

There are two ways to start an agent: by means of a VETkScript @new command, and by loading a Web page containing the agent.

From the moment a @new command is issued by the server, the following things happen:

- The agent and its window are assigned IDs, and an agent creation request is sent to the appropriate machine.
- The agent is instantiated as a Java object in the regular manner.
- The VCL init clause is executed, then the init_s script is executed.
- The agent starts handling messages.

Note that the execution of the @new action is asynchronous, and the script that started the agent continues executing right after the first step.

From the moment an agent is loaded within a Web page, the following things happen:

- The agent connects to the server and obtains an ID.
- The agent waits for all other agents on the page to obtain an ID, and these IDs become available to the agent as script variables.
- The VCL init clause is executed, then the init_s script is executed.
- The agent starts handling messages.

A.1.2 Agent exit

When an agent exits, or is deleted or disconnected, all the database information related to its ID is deleted as an atomic action. An agent is unregistered when:

- The connection of the server with the agent's virtual machine is broken.
- The agent exits using the VCL exit command, or throws an exception.
- The window the agent is contained in is closed by @wclose or by the user closing the window through the GUI.
- It is exited by the browser using the applet destroy() method.

A.1.3 Update notification

For VETkScript addition and deletion commands, @update +, @update -, @updateq +, and @updateq -, the update is issued immediately, as an atomic action. The difference of the database views before and after the action is determined, and results in update messages, sent to all concerned agents. VETkScript event update commands, @update . and @updateq ., are actually equivalent to two consecutive updates, an addition immediately followed by a deletion.

The content of the update may contain additions as well as deletions to one or more of the agent's views. When an agent receives an update, the following things happen:

- The agent's view results are updated accordingly.
- The VCL startupdate function is executed.
- All deletions from the original view results are handled by calling the corresponding handler, in undefined order.
- All additions to the view results are handled, in undefined order.
- The VCL endupdate function is executed.

A.2 VETkScript

Basically, VETkScript is just Jython with some extra commands. The only real change made to the Jython language is the replacement of the scoping marks. Instead of indentation, VETkScript uses the symbols '{{' and '}}' to indicate the start and end of a scope, resulting in a scoping syntax similar to C or Java. A second minor change is the addition of Java-like comments, '//' and '/* */'. Next to these minor changes, VETkScript introduces a number of commands, which are prefixed by an '@' symbol.

When a script is executed, the following variables are defined:

- **self** - the agent's ID.
- **self_window** - the agent's window ID.
- **any variables set through VCL execscript parameters.**
- **all variables defined in the script that started this agent through a @new command.** In other words, an agent inherits its variables from the agent that started it. There are no shared references, i.e. a deep copy is made of the variables.
- in case the agent is an applet, **any agent IDs of other applets running on the same page.** The variable names that the IDs are assigned to are given by the applets' *name* tags.
- **is_outdated** - a flag which indicates whether updates have been sent to the agent after the script was submitted for execution. So, if it is 1, the agent may have received information that might have outdated the script.

All variables remain defined during the agent's execution.

There are two kinds of command: actions, which have a different syntax depending on the kind of action, and functions, which take bracketed parameters, like regular functions. Both are described below.

A.2.1 Actions

Actions are the database actions and the creation of a new agent.

- **@new** <**id_var**> <**agenttype**> <**agentparameters**> - create new agent of type agenttype, store its ID in *id_var*, and store its window ID in *id_var_window*.

  The agent parameters have the format: **ListOf[** <**name**> **'='** <**value**> **]**.
- **@new** <**id_var**> - create new entity key and store its ID in *id_var*.
- **@addquery** <**view_name**> <**parameterlist**> **:** <**predicates**> - Couple a database query to the view of this agent called *view_name*.

  The parameter list is a comma-separated list of Jython expressions, in which any variables bound in the predicates may be used.

- **@doquery** <**parameterlist**> **:** <**predicates**> - Do immediate query and return the results as a list of tuples.
- **@update** <**action**> <**predicates**> - Add or delete predicates. action may be one of '+', (addition) '-', (deletion) or '.' (event, an addition followed by a deletion).
- **@updateq** <**action**> <**predicates_to_upd**> **:** <**query_predicates**> - Add or delete pattern. *predicates_to_upd* may contain variables defined in the query *query_predicates*. For each result of the query, the variables corresponding to the result are filled in in *predicates_to_upd*.

The predicates clauses are found as additions and deletions (indicating tuples to add and delete), and queries (indicating a Prolog-style search pattern). Each clause is a list of database tuples (see the database section). Each tuple parameter may be:

- A Jython expression, indicating a regular value.
- The 'any' pattern '_'. This indicates that any value matches at that position. May only be used in deletions and queries.
- A variable pattern: a tilde '~' followed by an identifier. This means that the value at that position will be bound to a variable of that name, and matched with other instances of that variable in other parameter positions. May only be used in queries.

In additions, each parameter may be prefixed by:

- The 'non-key' flag '*'. This means the parameter is not one of the 'database key' values that indicates unique identification of the tuple. Storing a tuple which has the same key values as an existing tuple will replace the old value, even if the non-key values are different.
- The 'agent' flag '@'. This means that the value (which should be a number) indicates an agent ID. If the value is an Agent, the flag is not needed since it is set automatically.

A.2.2 Functions

Beside the actions, there are some functions which take regular parameters as regular functions do.

- **@execute(scripttype, URL [, destination])** - execute HTML (scripttype = "html") or VETkScript (scripttype = "vs") script with given URL. For the case of HTML, destination indicates the name of the HTML frame to display the page.

- **@window(var_id [, x [, y [, gridx [, gridy]]]])** - Create new (not yet visible) window. *var_id* is a string indicating the name of the variable to assign the window ID to.
- **@wadd(parent, child)** - Add window as component inside other window. In case parent is root, display the window on the screen.
- **@wclose(window)** - Close window's top parent and all its children, and exit all agents contained within the window.

Note that the language is still lacking a '@delete' command to delete an agent. For now, we can use @wclose to delete an agent.

A.3 Vetk Data Constraint language (VDC)

The language enables specification of an entity-relationship model. One may define entities and relationships in the following way:

```
[ 'entity' | 'relation' ]  '('
      ListOf[ <attribute> | <entity> ':' <multiplicity> ]
')' '{'
      <values and events>
      <sets>
      <statemachines>
      <assertions>
'}'
```

The main distinction between entities and relations is that entities have a separate ID. An entity or relation may have attributes, and may have references to any number of entities, in which case the multiplicity of the reference is specified. Note that entities too may refer to other entities. The corresponding graphical notation for references in entities is the following: an arrowhead on the reference's arc points to its target. Entities and relations translate to the following tuples in the database:

```
<entityname>   (<ID>, <attribute/ref1>, <attr/ref2>, ...)
<relationname> (      <attribute/ref1>, <attr/ref2>, ...)
```

An entity or relation has a body, which specifies values (which are optional attributes), events, and sets. Values and events are denoted by an identifier, followed by an optional multiplicity, i.e.:

```
['value' | 'event' ] <identifier>
                - value or event with one attribute
['value' | 'event' ] <identifier> '/' <number>
                - value or event with number attributes
```

Multiple values and events may be specified in a single statement by separating them by commas. Values and events translate to the following tuples:

```
valuename(<entityID>, <attribute1>, <attribute2>, ... )
eventname(<entityID>, <attribute1>, <attribute2>, ... )
```

We may specify state machines inside entities. These are basic finite-state automatons, with the difference that a transition may be specified by two arbitrary assertions: one specifies an event, and the other a state condition. A transition occurs when the state condition is true and the event occurs (i.e. the condition specified in the event goes from false to true). A state automaton is specified by:

```
'statemachine' '{'
    SequenceOf[
        <sourcestate> '->' <destinationstate>
            '{' <event> '}'  [  '{' <condition> '}'  ]
    ]
'}'
```

The very first sourcestate specified is the begin state, which is entered as soon as the entity is created. sourcestate may be left out, in which case the same sourcestate as the previous transition is assumed. We may leave out the condition clause entirely indicating that it is not applicable, or we may specify an empty event clause to indicate that we do not specify an event.

Assertions may be specified inside an entity's or relation's body, in which case they are checked for each entity/relation separately, or as separate statements, indicating that they are global. Assertions are of the format:

```
'assert'   [ ListOf( <state> ) ]  '{'  <assertion>  '}'
```

In case the assertion is local, we may specify a state. The assertion only has to hold when the entity is in that state.

The events, conditions, and assertions are OCL-like truth expressions. The expressions have the following form:

```
  <setexpression>
| 'exists' '(' <setexpression> ')'
| 'forall' '(' <setexpression> ')'
| 'size' '(' <setname> ')' <comparison_op> <value>

|  <expr1>  [ 'and' | 'or' ]  <expr2>
|  'not' <expr>
```

With the *exists* and *forall* quantifiers, the specified *setexpression* is evaluated for each combination of elements in each of the sets mentioned in the expression. In case of

exists, the result is true as soon as one of the expressions results in *true*. In case of *forall*, the result is true if all of the expressions result in *true*. Specifying just a setexpression is equivalent to using an *exists* quantifier. The *size* operator counts the number of elements in a set. Finally, we may combine any of the above expressions using the Boolean operators *and*, *or*, and *not*.

Set expressions are expressions over query sets resulting in a Boolean value:

```
   <setname> '.' <fieldname>
|  <expr1>  <2-ary operator>  <expr2>
|  <1-ary operator>  <expr>
```

Basically, these are arbitrary VETkScript expressions in which individual fields of previously-declared sets may be addressed using the notation *setname.fieldname*.

A.3.1 VCL Structure

A VCL specification consists of zero or more agent specifications. An agent is specified by its name and the arguments it takes at startup, and its body:

```
'agent' <name> '('
    ListOf[  <arg_type> <arg_name>
                  [ '=' <arg_default_value> ]
          ]
')'  '{'
    <agent_body>
'}'
```

The argument types should be Java object types; primitive types are not allowed (i.e. use Integer instead of int). Any arguments that are not assigned values when an instance of the agent is created, are assigned the default values. The default default value is null. All arguments may be accessed from anywhere within the agent's body.

The arguments include 'regular' arguments as well as scripts. In the current version, scripts are simply arguments of type String. The variable IDs of scripts are usually suffixed by a '_s' (i.e. a script named myscript is typically called myscript_s). In future versions of VCL, scripts should be assigned a proper type and enable the parameters it expects to be defined.

An agent has several implicit arguments:

- **init_s** - the initialisation script.
- **width** - the width of its window, if applicable.
- **height** - the height of its window, if applicable.

The agent's body contains a number of clauses. A clause has the following format:

```
<clause_type> [ <additional_information> ] '{'
    <clause_body>
'}'
```

The clause body is a piece of regular Java code. There are two special commands for processing sets.

```
'take' '(' ListOf [ <java_type> <variable> ] ')'
        <expr_defining_a_set>

'foreach' '(' ListOf [ <java_type> <variable> ]  ';'
              <expr_defining_a_set>                  ')'
'{'  <body>  '}'
```

take takes one element out of a set (i.e. a tuple) and stores the respective elements of the tuple in the list of variables defined between brackets. These variables are from then on accessible. **foreach** enumerates the tuples in a set in a similar way: by assigning the elements of each tuple to the given list of variables. For both statements, one defines the set to be enumerated by means of a Java expression resulting in an object of type *Results* (such as the *results(...)* method).

Here is a list of the different clause types:

- **inline** - code to be included literally in the Java class to be generated. This can be used to define variables or regular internal methods.
- **init** - code to execute when agent initialises
- **startupdate** - code to execute at the beginning of an update (consisting of one or more elements of one or more views to be added or deleted).
- **endupdate** - code to execute when all elements of the update have been handled.
- **'handle'** <**view_id**> **'('** <**parameters**> **')'** - Define a database view *view_id*, and an input handler, which is the code to execute as reaction to an addition or deletion of the view's current result. Within the code, the booleans *is_addition* and *is_deletion* may be used to determine the nature of the update.
- **'handleinternal'** <**handler_id**> **'('** <**parameters**> **')'** - code to execute when an internal message arrives.

There is one special clause, the imports clause:

```
'imports'  '{'
    ListOf[  <java_module_import>  ]
'}'
```

This clause defines which Java modules should be imported for each agent. A default set of imports, to be used by all agents, may be defined as a separate clause, at the beginning of the specification. Additionally, an imports clause may be defined inside an agent, in which case it must occur at the beginning of an agent. This clause overrides the default imports.

A.3.2 Functions

The following Java methods may be used from within any clause body:

- **Results results(String view_id)** - Retrieve the current content of the given view.
- **void execscript(String scr [, String id1,Object value1 [, String id2,Object value2 [, String id3,Object value3]]])** - Assign values to the script variables named id1,id2,..., then execute script scr.
- **void execscript(String scr, String [] ids, Object [] values)** - Assign each of the variables with names as given in ids the values as given in values.
- **void exit()** - Exit successfully and silently.
- **void exit(String reason)** - Exit with error. May be invoked to exit on an unexpected situation. An error message is generated.
- **void activateVMMessages(String handler_id)** - Enables messages reporting the creation and deletion of other agents within the Java Virtual Machine that this agent runs in. The internal message handler called has the signature: *<handler_id>(Agent agent,Integer state)* with *state* being one of AgentCallbacks.ADD or AgentCallbacks.DEL.

Bibliography

Agarwal, R., Sinha, A. P., and Tanniru, M. (1996). The role of prior experience and task characteristics in object-oriented modeling: an empirical study. *International journal of human-computer studies*, 45:639–667.

Albesano, D., Baggia, P., Danieli, M., Gemello, R., Gerbino, E., and Rullent, C. (1997). Dialogos: A robust system for human-machine spoken dialogue on the telephone. In *Proc. ICASSP '97*, pages 1147–1150, Munich, Germany.

Alexander, H. (1990). *Formal methods in human-computer interaction*, chapter 9: Structuring dialogues using CSP. Cambridge university press.

Anastassakis, G., Panayiotopoulos, T., and Ritchings, T. (2001). Virtual agent societies with the mVITAL intelligent agent system. *Lecture Notes in Computer Science*, 2190:112–125.

Andernach, J. A. (1996). A machine learning approach to the classification and prediction of dialogue utterances. In *Proceedings of the Second International Conference on New Methods in Language Processing*, pages 98–109.

Andernach, J. A., Burgt, S. P. v., and Hoeven, G. F. v., editors (1995). *TWLT9: Corpus-based approaches to dialogue modelling*.

André, E., Graf, W., Mller, J., Profitlich, H.-J., Rist, T., and Wahlster, W. (1997). AiA: Adaptive communication assistant for effective infobahn access.

André, E., Rist, T., van Mulken, S., Klesen, M., and Baldes, S. (2000). The automated design of believable dialogue for animated presentation teams.

Androutsopoulos, I., Ritchie, G. D., and Thanisch, P. (1995). Natural language interfaces to databases - an introduction. *Natural Language Engineering 1:1*, pages 29–81.

Anonymous (2001). Step into your new browser. MIT technology review.

Atwood, M. E., Burns, B., Girgensohn, A., Lee, A., Turner, T., and Zimmermann, B. (1995). Prototyping considered dangerous. In *Proceedings of INTERACT'95: Fifth IFIP Conference on Human-Computer Interaction*, pages 179–184.

Baber, C. and Hone, K. S. (1993). Modelling error recovery and repair in automatic speech recognition. *International Journal of Man-Machine Studies*, 39(3):495–515.

Baltag, A. (1999). A logic of epistemic actions. In van der Hoek, W., Meyer, J.-J., and Witteveen, C., editors, *ESSLLI99 workshop: Foundations and applications of collective agent based systems (CABS)*.

Banavar, G., Doddapaneni, S., Miller, K., and Mukherjee, B. (1998). Rapidly building synchronous collaborative applications by direct manipulation. In *CSCW98: The 1998 ACM conference on computer supported cooperative work*.

Barr, A. and Tessler, S. (1996). Good programmers are hard to find: An alternative perspective on the immigration of engineers. *Stanford Computer Industry Project press release*.

Baskin, J. D. and John, B. E. (1998). Comparison of GOMS analysis methods. In *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems (Summary)*, volume 2 of *Late Breaking Results: Ubiquitous Usability Engineering*, pages 261–262.

Beazley, D. M. (1999). *Python Essential Reference*. New Riders Publishing.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Benysh, D. V. and Koubek, R. J. (1993). The implementation of knowledge structures in cognitive simulation environments. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 309–314.

Bernsen, N. O. (1996). Towards a tool for predicting speech functionality. *Free Speech Journal*, 1.

Bernsen, N. O., Dybkjaer, H., and Dybkjaer, L. (1998). *Designing interactive speech systems: from first ideas to user testing*. Springer Verlag.

Bhola, S., Banavar, G., and Ahamad, M. (1998). Responsiveness and consistency tradeoffs in interactive groupware. In *CSCW98: The 1998 ACM conference on computer supported cooperative work*.

Bjornson, R. (1992). *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, Department of Computer Science.

Blackwell, A. F. (2001). See what you need: Helping end-users to build abstractions. *Visual Languages and Computing*, 12(5):475–499.

Blackwell, A. F. and Green, T. R. G. (1999). Investment of attention as an analytic approach to cognitive dimensions. In *11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*.

Booch, G. (1994). *Object-oriented analysis and design, second edition*. Benjamin/Cummings.

Boussinot, F., Susini, J.-F., and Hazard, L. (1998). Distributed reactive machines. Technical Report 3376, INRIA, France.

Bouwman, A. G. G. (1998). Spoken dialog system evaluation and user-centered redesign with reliability measurements. Master's thesis, University of Twente, Deparment of Computer Science. February draft.

Braude, E. J. (2001). *Software Engineering: An Object-Oriented Perspective*. John Wiley & Sons Ltd.

Brazier, F., Cornelissen, F., Gustavsson, R., Jonker, C. M., Lindeberg, O., Polak, B., and Treur, J. (1998). Agents negotiating for load balancing of electricity use. In *Proceedings of the 18th international conference on distributed computing systems (ICDCS'98)*, pages 622–629.

Brazier, F., Dunin-Keplicz, B., Jennings, N., and Treur, J. (1995). Formal specification of multi-agent systems: a real-world case. In *First International Conference on Multiagent Systems*.

Brazier, F. M. T., Langen, P. H. G. v., J., T., Wijngaards, N. J. E., and M., W. (1994). Modelling a design task in DESIRE : the VT example. Technical Report IR-377, Faculteit der Wiskunde en Informatica, Vrije Universiteit, Netherlands.

Bretier, P. and Sadek, D. (1997). A rational agent as the Kernel of a cooperative spoken dialogue system: Implementing a logical theory of interaction. In Müller, J. P., Wooldridge, M. J., and Jennings, N. R., editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193 of *LNAI*, pages 189–204, Berlin. Springer Verlag.

Broersen, A. and Nijholt, A. (2002). Developing a virtual piano playing environment. In *Proceedings IEEE International Conference on Advanced Learning Technologies (ICALT 2002)*, pages 278–282.

Brooks, F. P. J. (1995). *The Mythical Man-Month, Anniversary Edition*. Addison-Wesley Publishing Co.

Brun, P. and Beaudouin-Lafon, M. (1995). A taxonomy and evaluation of formalisms for the specification of interactive systems. In *People and computers X: Proceedings of the HCI'95 conference*.

Busemann, S., Declerck, T., Diagne, A. K., Dini, L., Klein, J., and Schmeier, S. (1997). Natural language dialogue service for appointment scheduling agents. In *Fifth Conference on Applied Natural Language Processing*, pages 25–32.

Byrne, M. D., D.Wood, S., Noisukaviriya, P., Foley, J. D., and Kieras, D. (1994). Automating interface evaluation. In *Proc. of CHI-94*, pages 232–237, Boston, MA.

Calvary, G., Coutaz, J., and Nigay, L. (1997). From single-user architectural design to PAC*: a generic software architecture model for CSCW. In *CHI '97*.

Campos, J. C. and Harrison, M. D. (1997). Formally verifying interactive systems: A review. In Harrison, M. D. and Torres, J. C., editors, *Design, Specification and Verification of Interactive Systems '97*, Eurographics, pages 109–124, Wien. Springer-Verlag. Proceedings of the Eurographics Workshop in Granada, Spain, June 4 – 6, 1997.

Carey, R. and Bell, G. (1997). *The Annotated VRML 2.0 Reference Manual, 1st Edition*. Addison-Wesley.

Cartwright, M. and Shepperd, M. J. (2000). An empirical investigation of an object-oriented software system. *Software Engineering*, 26(8):786–796.

Cassell, J. (2001). Embodied conversational agents: Representation and intelligence in user interface. *AI Magazine*, 22(3):67–83.

Castelfranchi, C. (1991). No more cooperation, please! In search of the social structure of verbal interaction. In Orthony, A., Slack, J., and Stock, O., editors, *AI and Cognitive Science Perspectives on Communication*. Springer Verlag.

Chase, J. D., Hartson, H. R., Hix, D., Schulman, R. S., and Brandenburg, J. L. (1993). A model of behavioral techniques for representing user interface designs. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 861–866.

Chu-Carroll, J. and Brown, M. K. (1997). Tracking initiative in collaborative dialogue interactions. In *ACL '97/EACL '97: Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and of the 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 262–270.

Cohen, P. P. and Oviatt, S. L. (1995). The role of voice input for human-machine communication. *Proc. Natl. Acad. Sci. USA*, 92:9921–9927.

Cooper, A. (1999). *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How To Restore the Sanity*. SAMS.

Cooper, T. A. and Wogrin, N. (1988). *Rule-based programming with OPS5*. Morgan Kaufmann publishers.

Cortés, M. and Mishra, P. (1996). DWCPL: A programming language for describing collaborative work. In *CSCW96: The 1996 ACM conference on computer supported cooperative work*.

Cosquer, F. J. N., Veríssimo, P., Krakowiak, S., and Decloedt, L. (2000). Support for distributed CSCW applications. *Lecture Notes in Computer Science*, 1752:295–326.

Coutaz, J. (1997). PAC-ing the architecture of your user interface. In *Proceedings of the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 15–32. Springer, Addison-Wesley.

Crosby, M. E., Scholtz, J., and Wiedenbeck, S. (2002). The roles beacons play in comprehension for novice and expert programmers. In *Psychology of Programming workshop PPIG 2002*.

Dahlback, N., Reithinger, N., and Walker, M. A. (1997). *Standards for Dialogue Coding in Natural Language Processing: Report on the Dagstuhl-Seminar*. Dagstuhl.

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. (1995). The effect of inheritance on the maintainability of object-oriented software: an empirical study. In *Proceedings of the 1995 international conference on software maintenance*.

Danieli, M. and Gerbino, E. (1995). Metrics for evaluating dialogue strategies in a spoken language system. In *Proceedings of the 1995 AAAI Spring Symposium on Empirical Methods in Discourse Interpretation and Generation*, pages 34–39.

Date, C. J. (1997). *A guide to the SQL standard : a user's guide to the standard language SQL*. Addison-Wesley.

Davies, S. P. (1993). Expertise and display-based strategies in computer programming. In *People and computers VIII: proceedings of the HCI '93 conference*.

Dignum, F., Dunin-Keplicz, B., and Verbrugge, R. (1999). Dialogue in team formation: a formal approach. In van der Hoek, W., Meyer, J.-J., and Witteveen, C., editors, *ESSLLI99 workshop: Foundations and applications of collective agent based systems (CABS)*.

Dijkstra, E. W. (1968). Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148.

Dillman, D. A., Tortora, R. D., Conradt, J., and Bowker, D. (1998). Influence of plain vs. fancy design on response rates for web surveys. In *Proceedings of Survey Methods Section*. ASA Meeting.

Dillon, T. W., Norcio, A. F., and DeHaemer, M. J. (1993). Spoken language interaction: Effects of vocabulary size and experience on user efficiency and acceptability. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 140–145.

Dingel, J. (2002). Introduction to the formal specification and development of software systems. Lecture notes, available at: http://www.cs.queensu.ca/ cisc422/readings.html.

Dix, A., Finlay, J., Abowd, G., and Beale, R. (1998). *Human-computer interaction, second edition*. Prentice hall.

Douce, C. (2001). Long term comprehension of software systems: A methodology for study. In *Psychology of Programming workshop PPIG 2001*.

Doyle, P. and Hayes-Roth, B. (1997). An intelligent guide for virtual environments. In Johnson, W. L. and Hayes-Roth, B., editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 508–509, New York. ACM Press.

Dunsmore, A., Roper, M., and Wood, M. (2000). Object-oriented inspection in the face of delocalisation. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 467–476. ACM Press.

Eastgate, R. (2001). *The Structured Development of Virtual Environments: Enhancing Functionality and Interactivity*. PhD thesis, University of Nottingham.

Eckert, W., Levin, E., and Pieraccini, R. (1998). Automatic evaluation of spoken dialogue systems. In *TWLT13: Formal semantics and pragmatics of dialogue*, pages 99–110.

Ek, Å. (1997). Margrathea - a planning tool for environment adaptation. Usability test and evaluation. Master's thesis, Lund university, Sweden.

Encarnacão, M. (1997). *Concept and realization of intelligent user support in interactive graphics applications*. PhD thesis, Faculty of computer science, Eberhard-Karls-University, Tübingen.

Ericsson, K. A. and Simon, H. A. (1980). Verbal reports as data. *Psychological review*, 87(3).

Eugster, P., Guerraoui, R., and Sventek, J. (2000). Distributed asynchronous collections: abstractions for publish/subscribe interaction. Technical Report DSC/2000/003, École Polytechnique fédérale de Lausanne (EPFL), Communication Systems Department (DSC), Switzerland. Available at http://dscwww.epfl.ch/EN/publications/techrep.asp.

Evers, M. (1999). A case study on adaptability problems of the separation of user interface and application semantics. Technical Report TR99-14, University of Twente, Centre for Telematics and Information Technology (CTIT).

Evers, M. and Nijholt, A. (2000). Jacob - an animated instruction agent in virtual reality. In *Proceedings of the 3rd International Conference on Multimodal Interaction (ICMI 2000)*.

Frascina, T. and Steele, R. A. (1993). Task analysis in design of a human-computer interface for a ward based system. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 226–230.

Fraser, N. M. (1995). Quality standards for spoken language dialogue systems: a report on progress in EAGLES. In *ESCA workshop on spoken dialog systems: theories and applications*, pages 157–160.

Gabbard, J. and Hix, D. (1997). Taxonomy of usability characteristics in virtual environments, final report to the office of naval research. Technical report, Virginia Polytechnic Institute and State University.

Geerts, F. and Kuijpers, B. (2003). Deciding termination of query evaluation in transitive-closure logics for constraint databases. In *ICDT 2003*, pages 190–206.

Gibbon, D., Moore, R., and Winski, R. (1997). *Handbook of Standards and Resources for Spoken Language Systems*. Mouton de Gruyter, Berlin.

Gilmore, D. J. (1995). Interface design: have we got it wrong? In *Proceedings of IFIP INTERACT'95: Human-Computer Interaction*, pages 173–178.

Gobet, F. (1999). *Chess, Psychology of*. MIT Press.

Graesser, A. C., Person, N., and Harter, D. (2001). Teaching tactics and dialog in AutoTutor. *International Journal of Artificial Intelligence in Education*. in press.

Graham, T. N. (1995). *Declarative Development of Interactive Systems*. R. Oldenbourg Verlag.

Gray, W. D., John, B. E., Stuart, R., Lawrence, D., and Atwood, M. E. (1990). GOMS meets the phone company: Analytic modeling applied to real-world problems. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, Foundations: Cognitive Ergonomics, pages 29–34.

Green, T. (1990a). Programming languages as information structures. In *Psychology of programming (computers and people series)*, pages 118–137.

Green, T. R. G. (1989). *Cognitive dimensions of notations*, pages 443–460. Cambridge University Press.

Green, T. R. G. (1990b). *The nature of programming*, pages 21–44.

Green, T. R. G. (1997). Cognitive approaches to software comprehension: results, gaps and limitations. extended abstract. In *Proceedings of the workshop on Experimental Psychology in Software Comprehension Studies '97*.

Green, T. R. G. (1999). *Building and manipulating complex information structures: Issues in Prolog programming*, chapter 5.

Green, T. R. G. and Benyon, D. (1995). Displays as data structures: entity-relationship models of information artefacts. In *Proceedings of IFIP INTERACT'95: Human-Computer Interaction*, pages 55–60.

Green, T. R. G. and Benyon, D. (1996). The skull beneath the skin: entity-relationship models of information artifacts. *International Journal of Human-Computer Studies*, 44(6):801–828.

Green, T. R. G. and Petre, M. (1992). When Visual Programs are Harder to Read than Textual Programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*.

Greg Phillips, W. (1999). Architectures for synchronous groupware. Technical Report 1999-425, Department of Computing and Information Science, Queen's University, Kingston, Ontario.

Grice, H. P. (1975). Logic and conversation. In Cole, P. and Morgan, J. L., editors, *Syntax and Semantics: Vol. 3: Speech Acts*, pages 41–58. Academic Press, San Diego, CA.

Grieskamp, W. and Lepper, M. (2000). Using use cases in executable Z. In *3rd IEEE International Conference on Formal Engineering Methods*, pages 111–120.

Grolaux, D., Van Roy, P., and Vanderdonckt, J. (2001). QTk: An integrated model-based approach to designing executable user interfaces. In *Proceedings of DSVIS 2001*.

Gurr, C. A. (1994). Supporting formal reasoning for safety-critical systems. *High Integrity Systems*, 1(4):385–396.

Haan, G. d., Veer, G. C. v., and Vliet, J. C. v. (1991). Formal modelling techniques in human-computer interaction. *Acta Psychologica*, 78(1-3):27–67.

Hall, A. (1997). Do interactive systems need specifications? In Harrison, M. D. and Torres, J. C., editors, *Design, Specification and Verification of Interactive Systems '97*, Eurographics, pages 1–12, Wien. Springer-Verlag. Proceedings of the Eurographics Workshop in Granada, Spain, June 4 – 6, 1997.

Halse, M.-A. (1991). *IRIS Explorer User's Guide*. Silicon Graphics Inc.

Harel, D. and Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill.

Heeman, P. A. and Allen, J. F. (1997). Intonational boundaries, speech repairs and discourse markers: modeling spoken dialog. In *ACL '97/EACL '97: Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and of the 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 254–261.

Highsmith, J. and Cockburn, A. (2001). Agile software development: The business of innovation. *IEEE Computer*.

Hirschman, L. and Thompson, H. (1996). *Overview of Evaluation in Speech and Natural Language Processing*, pages 475–518. Cambridge University Press.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall, New York.

Holzmann, G. J. (1991). *Design and validation of computer protocols*. Prentice Hall.

Hone, K. S. and Baber, C. (1995). Using a simulation method to predict the transaction time effects of applying alternative levels of constraint to user utterances within speech interactive dialogues. In *ESCA workshop on spoken dialog systems: theories and applications*, pages 209–212.

Horrocks, I. (1998). *Constructing the User Interface with Statecharts*. Addison-Wesley.

Hussey, A. and Carrington, D. (1996a). Using Object-Z to compare the MVC and PAC architectures. In Roast, C. R. and Siddiqi, J. I., editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*.

Hussey, A. and Carrington, D. (1996b). Using Object-Z to specify a web browser interface. Technical Report TR96-06, Software Verification Research Centre, The University of Queensland.

Hussey, A. and Carrington, D. (1997a). An empirical study of formal user-interface design. Technical Report TR97-24, Software Verification Research Centre, The University of Queensland.

Hussey, A. and Carrington, D. (1997b). Specifying the UQ* editor user-interface with Object-Z. Technical Report TR97-02, Software Verification Research Centre, The University of Queensland.

Hussey, A., MacColl, I., and Carrington, D. (2001). Assessing usability from formal user-interface designs. In *13th Australian Software Engineering Conference (ASWEC'01)*, page 40.

Hussmann, H., Demuth, B., and Finger, F. (2000). Modular architecture for a toolset supporting OCL. In Evans, A., Kent, S., and Selic, B., editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer.

Isakowitz, T., Stohr, E. A., and Balasubramanian, P. (1995). RMM: A methodology for structured hypermedia design. *Communications of the ACM*, (8):34–44.

Issarny, V., Bellissard, L., Riveill, M., and Zarras, A. (2000). Component-based programming of distributed applications. *Lecture Notes in Computer Science*, 1752:327.

Iwadera, T., Ishizaki, M., and Morimoto, T. (1995). Recognizing an interactional structure and topics of task-oriented dialogues. In *ESCA workshop on spoken dialog systems: theories and applications*, pages 41–44.

Jameson, A. and Weis, T. (1995). How to juggle discourse obligations. In *Proceedings of the Symposium on Conceptual and Semantic Knowledge in Language Generation*, pages 171–185.

John, B. E. and Marks, S. J. (1997). Tracking the effectiveness of usability evaluation methods. *Behaviour and Information Technology*, 16(Issue 4–5):188–202.

John, B. E. and Vera, A. H. (1992). A GOMS analysis of a graphic, machine-paced, highly interactive task. In *Proc. of CHI-92*, pages 251–258, Monterey, CA.

Johnston, M., Cohen, P. R., McGee, D., Oviatt, S. L., Pittman, J. A., and Smith, I. (1997). Unification-based multimodal integration. In *ACL '97/EACL '97: Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and of the 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 281–288.

Jonker, C. M. and Treur, J. (1998a). Agent-based simulation of reactive, pro-active and social animal behaviour. In *Proceedings of the 11th international conference on industrial and engineering applications of AI and expert systems (IEA/AIE'98)*, volume 1, pages 584–595.

Jonker, C. M. and Treur, J. (1998b). Agent concepts and agent behaviour: an introduction. Part of the course handouts of the SIKS course on interactive and multi-agent systems, 30 november-4 december 1998.

Jönsson, A. (1993). *Dialogue management for natural language interfaces*. PhD thesis, Linköping University, department of Computer and Information Science.

Jorgensen, A. and Aboulafia, A. (1995). Perceptions of design rationale. In Nordby, K., Helmersen, P. H., d, D. J. G., and Arnesen, S. A., editors, *Proceedings of Human Computer Interaction: Interact '95*. Chapman and Hall: London.

Kaptelinin, V., Kuutti, K., and Bannon, L. (1995). Activity theory: Basic concepts and applications. *Lecture Notes in Computer Science*, 1015:189.

Keizer, S., op den Akker, R., and Nijholt, A. (2002). Dialogue act recognition with bayesian networks for dutch dialogues. In Jokinen, K. and McRoy, S., editors, *Proceedings 3rd SIGdial Workshop on Discourse and Dialogue*, pages 88–94.

Kelter, U., Monecke, M., and Schild, M. (2002). Do we need 'agile' software development tools? In *Net.ObjectDays (NODe) 2002*.

Khazaei, B. and Roast, C. (2001). The usability of formal specification representations. In *PPIG 2001: 13th Annual Workshop Programme*.

Kielmann, T. (1996). Designing a coordination model for open systems. In Ciancarini, P. and Hankin, C., editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 267–284. Springer-Verlag.

Kieras, D. (1996). *A Guide to GOMS Model Usability Evaluation Using NGOMSL*.

Kieras, D. E., Wood, S. D., and Meyer, D. E. (1997). Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. *ACM Transactions on Computer-Human Interaction*, 4(3):230–275.

Kitano, H. and Ess-Dykema, C. v. (1991). Toward a plan-based understanding model for mixed-initiative dialogues. In *29th Annual Meeting of the Association for Computational Linguistics*, pages 25–32.

Kitchenham, B. (1996). *Software metrics : measurement for software process improvement*. Oxford : NCC Blackwell.

Kitchenham, B. and Carn, R. (1990). *Research and practice: software design methods and tools*, pages 271–284.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C. (1994). Change impact identification in object oriented software maintenance. In *Proceedings of the 1994 international conference on software maintenance*.

Labrou, Y. and Finin, T. (1997). A proposal for a new KQML specification. Technical Report TR CS-97-03, University of Maryland Baltimore County.

Lambert, L. and Carberry, S. (1992). Modeling negotiation subdialogues. In *30th Annual Meeting of the Association for Computational Linguistics*, pages 193–200.

Lauesen, S. and Harning, M. B. (1993). Dialogue design through modified dataflow and data modelling. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 220–225.

Lester, J., Towns, S., Callaway, C., Voerman, J., and FitzGerald, P. (2000). *Deictic and Emotive Communication in Animated Pedagogical Agents*, chapter 5. MIT Press.

Levinson, S. C. (1987). Minimization and conversational inference. In Verschueren, J. and Bertucelli-Papi, M., editors, *The pragmatic perspective: selected papers from the 1985 International Pragmatics Conference*, pages 60–129. Benjamins.

Lewin, I. (1997). *3dt: User Manual (draft April 13 1997)*.

Li, D. and Muntz, R. (1998). COCA: Collaborative object coordination architecture. In *CSCW98: The 1998 ACM conference on computer supported cooperative work*.

Lim, K. Y. and Long, J. B. (1993). Structured notations for human factors specification of interactive systems. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 325–331.

Limbourg, Q., Pribeanu, C., and Vanderdonckt, J. (2001). Towards uniformed task models in a model-based approach. In *Proceedings of DSVIS 2001*, pages 164–182.

Markopoulos, P. (1997). *A compositional model for the formal specification of user interface software*. PhD thesis, University of London.

Marshall, L. and Webber, J. (2000). Gotos considered harmful and other programmers' taboos. In *Psychology of Programming workshop PPIG 2000*.

Martin, J.-C. (1997). Towards 'intelligent' cooperation between modalities: the example of multimodal interaction with a map. In *Proceedings of the IJCAI'97 workshop on Intelligent Multimodal Systems*.

McInnes, F. R., White, L. S., Foster, J. C., and Jack, M. A. (1995). An automated style checker for human-computer dialogue engineering. In *ESCA workshop on spoken dialog systems: theories and applications*, pages 149–152.

Medvidovic, N. and Taylor, R. N. (1997). A framework for classifying and comparing architecture description languages. *SIGSOFT Software Engineering Notes*, 22(6):60–76.

Meyer, B. (1985). On formalism in specifications. *IEEE Software*, 2(1):6–26.

Meyer, B. (1997). *Object-oriented software construction (2nd ed.)*. Prentice-Hall.

Meyer, J.-J. C. (1998). Intelligent agents, thema-intro en enige theoretische achtergronden. Course notes from SIKS 98.

Mezzanotte, M. and Paternó, F. (1996). Verification of properties of human-computer dialogues with an infinite number of states. In Roast, C. R. and Siddiqi, J. I., editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*.

Mills, K. L. (1996). An experimental evaluation of specification techniques for improving functional testing. *Systems Software*, pages 83–95.

Milner, A. J. R. G. (1980). *A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92*. Springer Verlag.

Milner, R. (1993). *The polyadic π-calculus: a tutorial*. Springer Verlag.

Minker, W. (1998). Evaluation methodologies for interactive speech systems. In *First International Conference on Language Resources and Evaluation (LREC)*, pages 198–206.

Moore, J. D. and Paris, C. L. (1989). Planning text for advisory dialogues. In *27th Annual Meeting of the Association for Computational Linguistics*, pages 203–211.

More, D. (1987). "goto considered harmful' considered harmful' considered harmful. *Communications of the ACM*, 30(5):351–352.

Moriyon, R., Szekely, P., and Neches, R. (1994). Automatic generation of help from interface design models. In *Proceedings of the ACM CHI 94*, pages 225–231.

Moulding, P. (2001). *PHP Black Book*. Coriolis.

Mullender, S. (1993). *Distributed Systems, 2nd edition*. Addison-Wesley.

Müller, J. P. (1996). *The design of intelligent agents: a layered approach*. Springer Verlag.

Munzner, T. and Burchard, P. (1995). Visualizing the structure of the World Wide Web in 3D hyperbolic space. In *Proceedings of the VRML 1995 Symposium*, pages 33–38. ACM Press.

Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., and Doane, P. (1997). The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365.

Nagao, K. and Takeuchi, A. (1994). Speech dialogue with facial displays: Multimodal human-computer conversation. In *Proceedings of ACL-94: the 32nd Annual Meeting of the Association for Computational Linguistics*.

Nardi, B. A. (1992). Studying context: A comparison of activity theory, situated action models, and distributed cognition. In *East-West International Conference on Human-Computer Interaction: Proceedings of the EWHCI'92*, pages 352–359. Missing from my collection.

Navarre, D., Palanque, P., Bastide, R., and Sy, O. (2001). Structuring interactive systems specifications for executability and prototypability. *Lecture Notes in Computer Science*, 1946:97–??

Nielsen, J. (1993). *Usability engineering*. Academic Press, New York.

Nielsen, J. (1995). Getting usability used. In Nordby, K., Helmersen, P. H., Gilmore, D. J., and Arnesen, S. A., editors, *Human-computer interaction: Interact '95*, pages 3–12. Chapman and Hall.

Nigay, L. and Coutaz, J. (1997). Software architecture modelling: bridging two worlds using ergonomics and software properties. In Palanque, P. and Paterno, F., editors, *Formal Methods in Human Computer Interaction*, chapter 3, pages 49–73. Springer Verlag.

Nijholt, A., Hessen, A. v., and Hulstijn, J. (1998). Speech and language interaction in a (virtual) cultural theatre. In *Natural language processing and industrial applications (NLP+IA 98) - Special accent on language learning*, pages 176–182.

Nijholt, A. and Hondorp, H. (2000). Towards communicating agents and avatars in virtual worlds. In de Sousa, A. and Torres, J. C., editors, *Proceedings EUROGRAPHICS 2000*, pages 91–95.

Norman, D. A. (1994). How might people interact with agents. *Communications of the ACM*, 37(7):68–71.

Object Management Group (2001). OMG unified modeling language specification, version 1.4. available at: http://www.omg.org/.

O'Brien, M. P., Shaft, T. M., and Buckley, J. (2001). An open-source analysis schema for identifying software comprehension processes. In *Psychology of Programming workshop PPIG 2001*.

Olson, J. R. and Olson, G. M. (1990). The growth of cognitive modeling in human-computer interaction since GOMS. *Human-Computer Interaction*, 5(2-3):221–265.

Pachet, F. (1995). On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4):19–24.

Palanque, P. and Bastide, R. (1995a). Formal specification and verification of CSCW using the interactive cooperative object formalism. In *Proceedings of the HCI'95 Conference on People and Computers X*, Formalism in HCI, pages 213–231.

Palanque, P. and Bastide, R. (1995b). Verification of an interactive software by analysis of its formal specification. In S., A., K., N., P., H., and D., G., editors, *Human-Computer Interaction, Interact'95*, pages 191–197. Chapman and Hall.

Palanque, P. and Bastide, R. (1996). A design life-cycle for the formal design of interactive systems. In Roast, C. R. and Siddiqi, J. I., editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*.

Palanque, P. A., Bastide, R., and Dourte, L. (1993). Contextual help for free with formal dialogue design. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 615–620.

Pasquini, A., Monfardini, G., Kaaniche, M., Mazet, C., Anderson, S., Embry, D., Rizzo, A., Veer, G. v. d., Sonneck, G., Ciciani, B., Laprie, J. C., Kanoun, K., Littlewood, B., Mortensen, U., Goerke, W., and Strigini, L. (1998). Lecture sheets and notes from the OLOS reliability and safety of human-computer systems summer school. Handouts.

Patterson, J. F., Day, M., and Kucan, J. (1996). Notification servers for synchronous groupware. In *Proceedings of ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, Protocols for Groupware, pages 122–129.

Petre, M. and Blackwell, A. F. (1999). Mental imagery in program design and visual programming. *International Journal of Human Computer Studies*, 51:7–30.

Philips (1998). *SpeechMania 2.0: Dialogue Description Language, developer's guide, overhead sheets*. Philips.

Pinheiro da Silva, P. (2000). User interface declarative models and development environments: A survey. In Palanque, P. and Paternò, F., editors, *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 207–226, Limerick, Ireland. Springer-Verlag.

Pinheiro da Silva, P. and Paton, N. W. (2000). UMLi: The unified modeling language for interactive applications. In Evans, A., Kent, S., and Selic, B., editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pages 117–132. Springer.

Polifroni, J., Seneff, S., Glass, J., and Hazen, T. J. (1998). Evaluation methodology for a telephone-based conversational system. In *First International Conference on Language Resources and Evaluation (LREC)*, pages 43–49.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., and Carey, T. (1994). *Human-Computer Interaction*. Addison-Wesley.

Purbrick, J. and Greenhalgh, C. (2000). Extending locales: Awareness management in MASSIVE-3. page 287.

Raley, J. B. (1996). Factors affecting the programming performance of computer science students. Master's thesis.

Ramduny, D., Dix, A., and Rodden, T. (1998). Exploring the design space for notification servers. In *Proceedings of ACM CSCW'98 Conference on Computer-Supported Cooperative Work*, Primitives for Building Flexibile Groupware Systems, pages 227–235.

Ramshaw, L. A. (1991). A three-level model for plan exploration. In *29th Annual Meeting of the Association for Computational Linguistics*, pages 39–46.

Reeves, J. W. (1996). Coping with exceptions. *the C++ Report*.

Reiner, M. (1995). Tools for collaborative learning in optics. In Beun, R., Baker, M., and Reiner, M., editors, *Dialogue and instruction. Modeling interaction in intelligent tutoring systems. Proceedings of the NATO Advanced Research Workshop on natural dialogue and interactive student modeling*, pages 137–155.

Richard, N., Codognet, P., and Grumbach, A. (2001). The InViWo toolkit: Describing autonomous virtual agents and avatars. *Lecture Notes in Computer Science*, 2190:195.

Rickel, J. and Johnson, W. L. (2000). *Task-Oriented Collaboration with Embodied Agents in Virtual Worlds*, chapter 4, pages 95–122. MIT Press.

Riel, A. J. (1996). *Object-oriented design heuristics*. Addison-Wesley.

Rising, L. and Janoff, N. S. (2000). The Scrum software development process for small teams. *IEEE software*, 17(4).

Roast, C. and Siddiqi, J. (1999). Using Z: the impact of specification upon quality. *PPIG Newsletter*, 24.

Robert, A., Chantemargue, F., and Courant, M. (1998). Grounding agents in EMud artificial worlds. In Heudin, J.-C., editor, *Proceedings of the 1st International Conference on Virtual Worlds (VW-98)*, volume 1434 of *LNAI*, pages 193–204, Berlin. Springer.

Roseman, M. and Greenberg, S. (1997). *Building Groupware with GroupKit*, pages 535–564. O'Reilly Press.

Rossi, G., Schwabe, D., and Lyardet, F. (1999). Web application models are more than conceptual models. In *Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*.

Rubin, F. (1987). 'goto considered harmful' considered harmful. *Communications of the ACM*, 30(3):195–196.

Russinovich, M. (1998). Windows NT and VMS: The rest of the story. *Windows NT Magazine*.

Ryan, M. D. and Sharkey, P. M. (1998). Distortion in distributed virtual environments. In Heudin, J.-C., editor, *Proceedings of the 1st International Conference on Virtual Worlds (VW-98)*, volume 1434 of *LNAI*, pages 42–48, Berlin. Springer.

Sadek, M. D., Bretier, P., and Panaget, F. (1997). ARTIMIS: Natural dialogue meets rational agency. In *international joint conference on artificial intelligence*, pages 1030–1035.

Sajaniemi, J. (2002). Visualizing roles of variables to novice programmers. In *Psychology of Programming workshop PPIG 2002*, pages 111–127.

Schank, P. K. and Linn, M. C. (1993). Supporting pascal programming with an on-line template library and case studies. *International journal of man-machine studies*, 38:1031–1048.

Schewe, K.-D. (2000). UML: A modern dinosaur? A critical analysis of the Unified Modelling Language. In Kangassalo, H., Jaakkola, H., and Kawaguchi, E., editors, *Proc. 10th European-Japanese Conference on Information Modelling and Knowledge Bases, Saariselkä (Finland), 2000*. IOS Press, Amsterdam.

Scholtz, J. (1993). A longitudinal study of transfer between programming language by experienced programmers. In *People and computers VIII: proceedings of the HCI '93 conference*.

Seo, J. and Kim, G. J. (2001). A structured approach to virtual reality system design. submitted to Presence 2001.

Shaw, E., Johnson, W. L., and Ganeshan, R. (1999). Pedagogical agents on the web. In *Proceedings of the Third Annual Conference on Autonomous Agents*, pages 283–290. ACM Press.

Sheppard, D. (1995). *An Introduction to Formal Specification With Z and Vdm*. McGraw-Hill.

Shneiderman, B. (1998). *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, 3rd edition.

Sikorski, T. and Allen, J. F. (1996). TRAINS-95 system evaluation. Technical Report TN96-3, University of Rochester, Computer Science Department.

Sime, M. E., Arblaster, A. T., and Green, T. R. G. (1977). Structuring the programmer's task. *Occupational psychology*, 50:205–216.

Smith, R. W. (1996). Initiative-dependent features of human-computer dialogs in a task-assistance domain. In *Energy Information Management VI: Conference Papers*, volume I: Computers in Engineering.

Smith, R. W. and Hipp, D. R. (1994). *Spoken Natural Language Dialog Systems: A Practical Approach*. Oxford University Press.

Smith, S., Bennett, K., and Boldyreff, C. (1995). Is maintenance ready for evolution? In *Proceedings of the 1995 interational conference on software maintenance*.

Soloway, E. and Ehrlich, K. (1989). Empirical studies of programming knowledge. In *Software reusability vol. II: applications and experience*, pages 235–267.

Sparck Jones, K. and Galliers, J. R. (1996). *Evaluation natural language processing systems, an analysis and review*. Springer Verlag.

Stein, A. and Maier, E. (1995). Structuring collaborative information-seeking dialogues. *Knowledge-Based Systems, Vol. 8 Iss. 2-3*, pages 82–93.

Stenning, K. and Gurr, C. (1996). Formal methods and human communication. In Roast, C. R. and Siddiqi, J. I., editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*.

Stent, A. J. and Allen, J. F. (1997). TRAINS-96 system evaluation. Technical Report TN97-1, University of Rochester, Computer Science Department.

Stork, A., Middlemass, J., and Long, J. (1995). Applying a structured method for usability engineering to domestic energy management user requirements: A successful case-study. In *Proceedings of the HCI'95 Conference on People and Computers X*, Task Analysis in Context, pages 367–385.

Stumm, M. (1993). A computer engineer's perception of software engineering. In *Proc. National Workshop on Software Engineering Education*.

Sutcliffe, R. J. (1987). *Introduction to Programming Using Modula-2 Columbus*. OH Merrill.

Taylor, R. N., Medvidovic, N., Anderson, K. M., E. Whitehead, J. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. (1996). A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406.

Topol, A. (2000). Immersion of XWindow applications into a 3D workbench. In *Proceedings of the ACM CHI'2000*.

Trafton, J. G., Wauchope, K., and Stroup, J. (1997). Errors and usability of natural language in a multimodal system. In *Proceedings of the IJCAI'97 workshop on Intelligent Multimodal Systems*.

Tullis, T. S. (1993). Is user interface design just common sense? In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 9–14.

Turk, D., France, R., and Rumpe, B. (2002). Limitations of agile software processes. In *Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002)*, Alghero, Sardinia, Italy.

van Dijk, B., op den Akker, R., Nijholt, A., and Zwiers, J. (2001). Navigation assistance in virtual worlds. In *Informing Science Conference*.

van Eijk, R., de Boer, F., van der Hoek, W., and Meyer, J.-J. (1999). Operational semantics for agent communication languages. In van der Hoek, W., Meyer, J.-J., and Witteveen, C., editors, *ESSLLI99 workshop: Foundations and applications of collective agent based systems (CABS)*.

van Schooten, B. (1999). Building a framework for developing interaction models: Overview of current research on dialogue and interactive systems. Technical Report TR-CTIT-99-04, University of Twente, Centre for Telematics and Information Technology.

van Schooten, B. (2000a). Process and agent based modelling techniques for dialogue systems and virtual environments. Technical Report TR-CTIT-00-04, University of Twente, Centre for Telematics and Information Technology. Available at http://wwwhome.cs.utwente.nl/˜schooten/.

van Schooten, B. (2000b). A specification technique for building interface agents in a web environment. In *TWLT17: Learning to Behave, workshop I: Interacting Agents*.

van Schooten, B. (2001). Structuring distributed virtual environments using a relational database model. Technical Report GIST G2001-1 (Early Proceedings of DSVIS 2001), Dept. of Computing Science, University of Glasgow, Scotland.

van Schooten, B. (2002). Using declarative constraints to specify the data model of a multi-user application. Early Proceedings of DSVIS 2002, Rostock, Germany.

van Schooten, B., Donk, O., and Zwiers, J. (1999). Modelling interaction in virtual environments using process algebra. In *TWLT15: Interactions in Virtual Worlds*.

Veer, G. C. v. d., Hoeve, M., and Lenting, B. F. (1996a). Modeling complex work systems - method meets reality. In *Proceedings of the Eight European Conference on Cognitive Ergonomics*, pages 115–120.

Veer, G. C. v. d. and Lenting, B. F. (1995). *Cognitive Ergonomie en MCI, lecture notes from the 1995 student course*. University of Twente, faculty of WMW, department of ergonomics.

Veer, G. C. v. d., Lenting, B. F., and Bergevoet, B. A. J. (1996b). GTA: Groupware task analysis - modeling complexity. *Acta Psychologica*, 91:297–322.

Veer, G. v. d., Mast, C. v. d., Jonker, C., Braspenning, P. J., Wiesman, F., Meyer, J., Poutre, L., and Weigand, H. (1998). Miscellaneous sheets from the 1998 SIKS course on interactive systems and multi-agent systems. Part of the course handouts of the SIKS course on interactive and multi-agent systems, 30 november-4 december 1998.

Vosinakis, S. and Panayiotopoulos, T. (2001). SimHuman: A platform for real-time virtual agents with planning capabilities. *Lecture Notes in Computer Science*, 2190:210.

Wahlster, W., Reithinger, N., and Blocher, A. (2001). Smartkom: Towards multimodal dialogues with anthropomorphic interface agents. In *MTI Status Conference*.

Walker, M., Hindle, D., Fromer, J., Di Fabbrizio, G., and Mestel, C. (1997a). Evaluating competing agent strategies for a voice email agent. In *EUROSPEECH97: Proceedings of the European Conference on Speech Communication and Technology*.

Walker, M. A. (1996). Limited attention and discourse structure. *Computational Linguistics*, 22-2.

Walker, M. A., Litman, D. J., Kamm, C. A., and Abella, A. (1997b). PARADISE: a framework for evaluating spoken dialogue agents. *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*.

Watt, S. N. K. (1997). Artificial societies and psychological agents. In *Software agents and soft computing*, pages 27–41.

Wauchope, K. (1996). Multimodal interaction with a map-based simulation system. Technical Report AIC-96-027, Naval Research Laboratory, Washington, DC.

Weinberg, G. M. (1971). *The Psychology of Computer Programming*. Van Nostrand Reinhold.

Wooldridge, M. (1998). Verifiable semantics for agent communication languages. In Demazeau, Y., editor, *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS 98)*. IEEE Press.

Wooldridge, M. (1999). *Intelligent Agents*, chapter 1. MIT Press.

Wright, P., Merriam, N., and Fields, B. (1997). From formal models to empirical evaluation and back again. In Palanque, P. and Paterno, F., editors, *Formal Methods in Human Computer Interaction*, chapter 14, pages 283–313. Springer Verlag.

Zhang, J. and Norman, D. A. (1994). Representations in distributed cognitive tasks. *Cognitive Science*, 18:87–122.

Zukowski, J. (2002). *Mastering Java 2, J2SE 1.4*. Sybex.

Summary

This thesis concerns the issues involved in the development of virtual environments (VEs). VEs are more than virtual reality. We identify four main characteristics of them: graphical interaction, multimodality, interface agents, and multi-user. These characteristics are illustrated with an overview of different classes of VE-like applications, and a number of state-of-the-art VEs.

To further define the topic of research, we propose a general framework for VE systems development, in which we identify five major classes of development tools: methodology, guidelines, design specification, analysis, and development environments. Of each, we give an overview of existing best practices.

We chose to emphasise design specification, that is, specifications that describe the system under development. We identify the place of such specifications within our development framework, and the shape that they take. We assess a number of specification languages. These were chosen according to both applicability to and actual use within the domain of VEs. The languages we cover include languages based on predicate logic, temporal logic, and intentional logic, production rule systems, structure-based models such as structure diagrams, dataflows, architecture description languages, constraint programming, and control flow based models such as statecharts, process algebras, and Petri nets.

The languages were assessed with help of four general suitability criteria: expressiveness, well-definedness, cognitive tractability, and computational tractability. In particular the notions of well-definedness (the coupling with the actual system under development) and computational tractability (the possibilities that a computer has to process the specifications) are emphasised, answering to some of the issues pointed out by some of the modern software development movements, such as agile development.

We argue that it is desirable to use multiple languages for design specification. With help of our language assessment, we develop a new specification technique (i.e. a set of interrelated specification languages) for specifying the executable system, abstract models of the system, and constraints that it should conform to.

The technique is centred around system architecture. Like typical user interface (UI) architecture models, it emphasises modelling of the structure of the UI. Unlike these models, it is centred around structure diagrams and a shared database, rather than encapsulation or interfaces. This leads to a comprehensive component and shared data

structure, which is useful for systems which need to share complex information such as multi-user or multimodal or context-aware UIs. The shared data is specified by a set of correctness constraints, which combine structure specifications with logic-based constraints. The technique is also component-based: components are specified independently of their coupling with a specific system. A component is coupled by specifying database subscriptions that define the component's input, and output scripts that transform the component's output into database actions.

The specification technique is illustrated by some examples, showing the feasibility of the concepts behind it. It is argued to be especially useful for: GUIs, UIs with complex structure, limited multimodal UIs, limited interface agents, and multi-user UIs. Multimodality is limited in the sense that no multimodal interpretation facilities are provided, nor is input from special devices, such as speech input. Interface agents are limited in the sense that no special behaviour specification facilities, such as a specialised language, is provided.

We conclude with a list of positive and negative points of the current technique, some possible improvements, and some future directions, as regards a first prototype that has been built of a 3D graphics and animation model, and possible options for cooperative behaviour specification of interface agents.

Samenvatting

Dit proefschrift gaat over de problematiek van de ontwikkeling van virtuele omgevingen (VE's). VE's zijn meer dan virtual reality. We identificeren vier hoofdeigenschappen: grafische interactie, multimodaliteit, interface agents, en meerdere gebruikers. We illustreren deze vier eigenschappen d.m.v. een overzicht van verschillende klasses van VE-achtige softwareapplicaties, en een aantal *state-of-the-art* VE's.

Om ons onderzoeksgebied verder te verhelderen stellen we een algemeen raamwerk voor voor het ontwikkelen van VE-software. Hierbij identificeren we vijf belangrijke klasses van ontwikkelgereedschap: methodologie, richtlijnen, ontwerpspecificatie, analyse, en ontwikkelomgevingen. Voor elk van deze geven we een overzicht van de bestaande praktijken die het best werken.

We kiezen ervoor om de nadruk te leggen op ontwerpspecificatie, d.w.z. specificaties die het systeem dat ontwikkeld wordt beschrijven. We identificeren de plaats van zulke specificaties in ons ontwikkel-raamwerk, en de vorm die ze aannemen. We bekijken een aantal specificatietalen. Deze zijn gekozen aan de hand van zowel de toepasbaarheid voor en gebruik in het domein van de VE's. De talen die we behandelen omvatten talen gebaseerd op predicaatlogica, temporele logica, en intentionele logica, productieregel-systemen, structuurgebaseerde modellen zoals structuurdiagrammen, dataflows, en architectuursbeschrijvingstalen, en *controlflow*-modellen zoals statecharts, procesalgebra's en Petri-netten.

De talen zijn nader bekeken met behulp van vier algemene geschiktheidscriteria: expressiviteit, welgedefinieerdheid, cognitieve trekbaarheid, en computationele trekbaarheid. Vooral de concepten welgedefinieerdheid (het verband met het uiteindelijke systeem dat ontwikkeld wordt) en computationele trekbaarheid (de mogelijkheden die een computer krijgt om de specificaties te verwerken) worden zo benadrukt, welk een antwoord geeft op een deel van de problematiek die aangegeven is door sommige moderne softwareontwikkelingspraktijken, zoals *agile* ontwikkeltechnieken.

We beargumenteren dat het nodig is om meerdere talen te gebruiken voor de ontwerpspecificatie. Met behulp van ons overzicht van talen ontwikkelen we een nieuwe specificatietechniek (d.w.z. een aantal aan elkaar gerelateerde specificatietalen) voor het specificeren van zowel het executeerbare systeem, abstracte modellen van het systeem, en eisen waar het systeem aan moet voldoen.

De techniek heeft als centraal punt de systeemarchitectuur. Zoals veel

gebruikersinterface-architectuurmodellen legt het de nadruk op het modelleren van de structuur van de gebruikersinterface (of user interface, UI). In tegenstelling tot zulke modellen maakt het gebruik van structuurdiagrammen en een gedeelde database (*shared database*), in plaats van encapsulatie en interfaces. Dit leidt tot een inzichtelijke component-structuur en gedeelde datastructuur, welke nuttig is voor systemen die complexe data moeten communiceren, zoals multi-user systemen en multimodale en context-gevoelige UI's. De gedeelde database wordt gespecificeerd door een verzameling correctheidseisen, die een combinatie zijn van structuurspecificaties en logische specificaties. De techniek is ook component-gebaseerd: de componenten worden onafhankelijk van hun integratie in een specifiek systeem gespecificeerd. Een component wordt dan geintegreerd door het specificeren van *database subscriptions* die de invoer van het component bepalen, en door uitvoer-scripts, die de uitvoer van een component verwerken tot database-acties.

De specificatietechniek wordt geïllustreerd d.m.v. enkele voorbeelden, die aantonen dat de concepten erachter inderdaad zinvol zijn. We concluderen dat het vooral nuttig is voor: grafische UI's, UI's met een complexe structuur, multimodale en contextgevoelige UI's, interface agents, en multi-user UI's. Ondersteuning van multimodaliteit is beperkt omdat er geen standaardfaciliteiten geleverd worden voor multimodale interpretatie, en ook niet voor invoer van speciale invoermiddelen zoals spraakinvoer. Interface agents worden ook beperkt ondersteund in die zin, dat er geen speciale gedragsspecificatiefaciliteiten geleverd zijn, zoals een gedragsspecificatietaal.

We sluiten de tekst af met een lijst van positieve en negatieve punten van de huidige techniek, een aantal mogelijke verbeteringen, en een aantal richtingen voor verder onderzoek, wat betreft een eerste prototype wat ontwikkeld is van een model voor 3D graphics en animatie, en mogelijke opties voor de specificatie van cooperatief gedrag van interface agents.

SIKS Dissertatiereeks

====
1998
====

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

====
1999
====

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

====
2000
====

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU)

ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

====
2001
====

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice
BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management:
The Role of Mental Models in Business Systems Design

====
2002
====

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic
Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of
the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive
Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative
E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy
Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational
Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and
Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow
Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

====
2003
====

2003-01 Heiner Stuckenschmidt (VU)

Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments