INVESTIGATION OF UIMS ARCHITECTURES
FOR THE DEVELOPMENT OF USER INTERFACES
CONCURRENTLY ACCESSIBLE BY BLIND
AND SIGHTED USERS

Anthony Savidis¹, Constantine Stephanidis²

Abstract

The proliferation of Graphical User Interfaces (GUIs) has presented several serious accessibility problems to blind people. Existing approaches to overcome such problems are customised to specific environments and are restrictive in their scope. Moreover, they do not take into consideration the specific dialogue requirements of the various graphical applications. It is argued that in order to resolve such existing problems and also to ensure that GUIs in the future will be made accessible to blind people, an appropriate methodology for the construction of interface development tools is required which will take in consideration and efficiently integrate both blind and sighted user requirements. The domain of User Interface Management Systems is investigated as the most appropriate epistemic basis and the design of a suitable interface development architecture is structured for the generation of Human-Computer dialogues that are concurrently accessible by blind and sighted people.

E-mail¹ as@csi.forth.gr, E-mail² cs@csi.forth.gr
Contents

Executive Summary 1

1. Introduction 2
   1.1. Outline of the proposed approach 3
   1.2. Anticipated impact 6
   1.3. Related work 6

2. Modelling framework of investigation 8
   2.1. Models of Human-Computer interaction 8
   2.2. Mapping UI models to implemented interactive systems - UIMS models
       2.2.1. Traditional Seeheim model 13
       2.2.2. The Arch model 14
   2.3. Discussion 16

3. Formal specification methods of Human-Computer dialogue 17
   3.1. Quantitative classification 17
   3.2. Qualitative classification 20
   3.3. Further discussion
       3.3.1. User modelling facilities 21
       3.3.2. Efficient on-line help handling 22
       3.3.3. Critique 23

4. The proposed UIMS-based architectural framework 25
   4.1. Terminology 25
   4.2. Principal functional requirements 26
   4.3. Conceptual view of the analytic design
       4.3.1. Multilevel structuring of the dialogue specification scenario
               4.3.1.1. Dialogue description and notational constructive tools 31
               4.3.1.2. Modelling morphological representation of application oriented information 35
               4.3.1.3. Overview - structure and relations 39
       4.3.2. Integrating and interfacing lexical technologies 42
       4.3.3. Enroling the application component 43
       4.3.4. The proposed run-time architectural scenario 43
       4.3.5. An example of a Dual Interface 46
   4.4. Analytic system design
       4.4.1. Conventional programming constructs 48
       4.4.2. Dependency schemes and constraint satisfaction 53
       4.4.3. Methods for objects & handlers for communication events 61
       4.4.4. Virtual object genesis 68
       4.4.5. Virtual object composition 72
       4.4.6. Dialogue agents 75
       4.4.7. Instantiation logic for virtual interaction objects 85
       4.4.8. Main dialogue control procedure for the higher-level 89

5. Summary, discussion and conclusions 89

References 91
Acknowledgements

The work reported here, has been carried out at ICS-FORTH in the context of the GUIB project, which is partially funded by the TIDE Pilot Action Programme of the Commission of the European Communities (DG XIII). The partners of this consortium are: IROE-CNR, Italy (Prime Contractor); F H Papenmeir GmbH & Co KG, Germany; IFI-University of Stuttgart, Germany; Institute of Computer Science-FORTH, Greece; RNIB, England; Institute of Telecommunications-TUB, Germany; Department of Computer Science-FUB, Germany; Vrije Universiteit Brussel, Belgium; VTT, Finland.
Executive Summary

As graphical User Interfaces become progressively more complex, in terms of the visual structures communicated to the user, conventional methods of providing accessibility to blind users based on adaptations at the lexical level tend to become practically inapplicable. Consequently, it becomes necessary to define an appropriate and efficient technological ground which would ensure continued access of future computer-based applications by blind users. Moreover, it is argued that in order to achieve the most promising results, a suitable “integrational” approach is required, with respect to both blind and sighted users; The proposed approach would enable the development of appropriate tools facilitating the uniform and integrated construction of User Interfaces accessible concurrently by blind and sighted users. This report presents the detailed results of an investigation which follows the above principles and concerns the definition of a novel interface development architecture which meets the specified requirements. The undertaken investigation has addressed the following interrelated areas:

a) General architectural design, which deals with the specification of an appropriate functional structuring of the software components which realise the User Interface management procedure.
b) Dialogue specification methodology, which concerns the design of a suitable formal technique, which would allow the unified higher-level interface description.
c) Run-time management scenarios, related to the design of efficient run-time algorithms for dialogue control, following the defined specification formalism.

This innovative approach aims to provide the necessary stimulus to developers of future interface construction tools, so that they can incorporate at an early design stage the various suggested technical architectural features. However, it is considered very important to conduct this investigation by taking into account the numerous existing and widely adopted theoretical methods and techniques and by revisiting, where appropriate, the various functional aspects according to the given additional requirements.

The structure of this report follows an incremental exposition of the analytic and synthetic aspects of the investigation effort that has been carried out. Section 1 provides an outline of the proposed approach and a brief discussion concerning related work in this field. Sections 2 and 3, concern an in depth analysis and discussion of the adopted theoretical basis (models of Human-Computer dialogue, relevant formal specification techniques and interface development architectures). Section 4, provides a detailed discussion which addresses the main design strategies and related key-issues in accordance to an analytic description of the proposed architectural design; various implementational details are also considered. Finally, in Section 5, a technical summary of the overall proposed approach is presented and certain key conclusions are drawn, within the context of the fundamental goals of the on-going work.
1. Introduction

For more than a decade now, blind people have been given the possibility to access text-based applications running on microcomputer systems, including access to a variety of information sources and systems, by using alternative, non-visual presentation techniques (based on speech and / or Braille). However, the emergence of Graphical User Interfaces (GUIs) and graphics-based applications have imposed enormous new constraints on blind people and current efforts to enable such accessibility are inherently restrictive in their scope. Moreover, the continued evolution of User Interface technology towards more visually oriented concepts and 3D representations, is expected to further limit the scope of currently adopted methodologies which primarily aim to achieve accessibility of graphical environments by blind users, based on adaptations at the lexical level (lexical adaptations).

Even with the application of very sophisticated run-time filtering, there are certain situations where direct reproduction of information to non-visual form is not feasible. More specifically, when information is conveyed in various pictorial forms (i.e. images, tabular data representations, bar-charts, diagrams, illustrations, artistic scenes, etc) no specific information is actually incorporated referring
to the represented semantics. Currently, it is not possible to achieve practically automatic deduction of the internal semantics exclusively from pure geometrical descriptions (i.e. image understanding methods). Moreover, domain oriented knowledge is required in order to distinguish whether common symbolisms correspond to differentiated semantics (e.g. resolving presentational ambiguity). It is also claimed that the designer of non-visual dialogues should be the one ultimately responsible to decide the forms and means of communicating efficiently information to the blind user, in given circumstances, as opposed to the existing solutions where the significant role of the designer for non-visual dialogues is practically neglected.

Considering that the trend of User Interface technology is directed towards interaction methods aiming to enable maximum exploitation of the human visual information processing capability, such as virtual reality and 3D representations (i.e. visual reality), it is expected that application of techniques based on lexical adaptations will eventually become unrealistic or meaningless. In [MACK91], [CARD91], [ROBE91] examples of implemented innovative interaction methods are demonstrated which are based on highly visual world metaphors. It is evident that reproduction of similar dialogues in a non-visual form by involving lexical level issues is not appropriate. Nevertheless, it should be noted that the methodological basis of such existing approaches is centred around the visual User Interface construction techniques and human factors considerations. In part, the cause of the present situation is the lack of the necessary technological ground for supporting non-visual User Interfaces within an appropriate and complete framework, corresponding to their visual counterparts (i.e. currently, no implemented analogy of a non-visual toolkit exists). A general and systematic methodology is required, to encourage progress on the development of non-visual interaction methods in accordance to an efficient framework which would facilitate further utilisation of such non-visual interaction techniques for the implementation of non-visual user-computer dialogue, and also would allow (where required) the efficient integration with particular visual dialogue techniques. In this way, the role of the designer of non-visual dialogues can be effectively supported.

1.1. Outline of the Proposed Approach

Today, there is a significantly limited ground for supporting efficiently the construction of non-visual User Interfaces, in comparison to their visual counterparts. Initially, most efforts were devoted on the general problem of enabling accessibility; consequently, the role of the designer for non-visual dialogues has been progressively declined by considering traditional issues related to good interface design being of minor importance. One of the main technical objectives of this task is to specify the appropriate technological tools which would allow higher level (i.e. without requiring programming efforts) development of such shared dialogues by efficiently supporting the role of dialogue designer, both for visual and non-visual interaction. Additionally, it has been considered critical to ensure that the designed technology would be extensible and portable in order to allow modular adaptations according to the various future technological trends, and also to enable implementation on various platforms and environments. In this framework, User Interface Management Systems (UIMS), which contributed effectively in the past for the accomplishment of fast prototyping and development cycles for visual interfaces, have been considered as the most appropriate technological basis. In this framework, the various relevant technical issues will be appropriately addressed.

The proposed approach is based on the conceptual separation of the various software modules, which constitute an interactive system, into two different distinct layers (see figure 1.1.1.): the interface specific layer (I/O) and the application specific layer (Process). Roughly speaking, the proposed framework should facilitate the provision of a common application layer with two appropriate interface layers. The development methodology will be based on the automatic generation of the implementation modules according to higher-level specifications. Also, suitable notational techniques will be specified which will allow the designer to efficiently capture and easily express the various commonalities in terms of visual and non-visual dialogues for a specific application.
For instance, information might be conveyed to various media (various special I/O devices may be employed for the provision of a non-visual physical level) but with the provision of a uniform access model. The different output media broaden seriously the output communication channel and enable a more direct understanding of information. Combination of visual and non-visual multimedia output devices and techniques such as:

- raster displays
- tactile displays
- thermal displays
- auditory displays
- video
- pictures
- animation
- speech and sounds
- 2D and 3D graphics
- force feedback

The provision of multimedia input is also critical; for example:

- gestures
- speech input
- 2D and 3D pointing.

In order to support multimedia information manipulation, it is important to study some of the structural principles of multimedia interfaces [BLAT90]. These principles may be aesthetic, ergonomic, or based on any other systematic approach. At the present time, composition of media and modality is intuitive, based on feelings and experiences. The choice of devices is most likely determined by what is accessible at a lowest cost. The goals for an understanding of composition of multimedia methods are in general:

- Effectiveness
- Expressiveness
- Naturalness
- Accessibility

In Figure 1.1.2., two different policies concerning the division of display surface for visual interfaces is presented. In principle, adaptation-oriented techniques would reproduce such a picture to a non-visual form. However, the underlying desk-top metaphor should be primarily investigated in order to decide when and how such interactive structures are meaningful for blind users. In the proposed approach, the flexible construction of dialogues for blind users is supported, without requiring the introduction of visually oriented dialogue primitives.
One of the ultimate goals driving this investigation was to provide an architectural framework capable of supporting dialogues accessible concurrently by sighted and blind users. The technical perspective of such an approach was mainly directed from the longer-term objective of supplying efficient development building-blocks for the generation of interfaces having, primarily, such important properties. Consequently, this research effort is practically considering the longer-term aspects of accessibility as viewed within the GUIB project. In the short-term, specific implementations and demonstrators are built in order to provide sophisticated adaptations for two appropriately selected graphical environments with the broadest commercial acceptance (MS-WINDOWSTM and X-WINDOW SYSTEM). Moreover, various alternative methods for non-visual interaction in a graphical environment and related usability evaluation methodologies have been studied. Also, the various relevant perceptual / cognitive issues have been primarily addressed. This short-term approach aims to provide concrete results for the selected environments through an integrated adaptation-oriented methodology which aims to supply the graphical environment concurrently to a suitable non-visual form. Moreover, such practical scenarios provide an appropriate background for the longer-term approach, which in principle is characterised by a wider scope (quantitatively and qualitatively) of applicability.
1.2. Anticipated Impact

Currently, a plethora of development technologies is available in the domain of graphical man-machine communication and the vast majority of interface developers employ such instrumental support. However, none of the existing tools incorporate built-in facilities which would enable accessibility of generated interfaces by blind users, either partially or totally. Also, it is argued that the underlying architectural philosophy of such existing tools does not allow practical extensions/adaptations which would support appropriately the construction of non-visual interfaces. Consequently, one could characterise the field of non-visual Human-Computer communication as the domain of missing methodologies and architectures; more sophisticated approaches than the traditional adaptation oriented strategies and ad-hoc solutions are required.

Clearly, accessibility constraints are mainly due to imperfections of the utilised underlying development methods. Assuming that software developers normally prefer widely available graphical environments and industry standards, a solution tight to a specific platform is unlikely to attract broad industrial interest. In the framework of our investigation, a novel strategy for developing interface builders has been formulated allowing the implementation of UIMS which would support the production of interfaces concurrently accessible by blind and sighted users. The proposed methodology is considered as a significant contribution to the scientific body of knowledge in the field and aims to practically influence future trends in the domain of User Interface software technology. It should be noted that the proposed theoretical framework has been provided with informal proofs, by means of practical implementational paradigms, which logically demonstrate its technical feasibility and realistic applicability.

Within the field of development tools, UIMS increasingly gain commercial acceptance and it is expected that they will dominate (in this specific domain) the market in the near future. Hence, the capabilities and properties of produced User Interfaces will depend on the efficiency of the particular UIMS-based tool employed. Consequently, the present time is considered as the most critical (appropriate) chronological point for providing the necessary stimulus to the community of UIMS developers through the introduction of such a multi-platform architectural framework. Such policy aims to modularly establish a safe bridge between visual and non-visual interfaces which will further enable integration of blind and sighted users in the working environment, and will also ensure continued access on future computer-based application due to the powerful properties of the supporting development tools.

1.3. Related Work

Significant work has been carried out in the recent past, in order to enable accessibility by blind users of interactive applications developed for character based terminals (i.e. programs running in text-mode for MS-DOS™ based personal computers). These systems are usually called screen readers and the vast majority is based on an appropriate resident program which drives additional peripheral devices (e.g. speech synthesizer, tactile module usually based on Braille, etc). Typically, such a resident program will reconstruct internally the visual display by scanning the video buffer and/or trapping specific interrupts for character oriented output functions. Also, appropriate interactive facilities are provided to allow reviewing of extracted screen contents by a blind user. IBM SCREEN READER™ (SRD) [ADAM89] is a well known system, that can operate with a large variety of speech synthesizers and is also available for the main European national languages. Another example is the PARLA™ system (reviewed in [GUIB92]), one of the most popular screen readers for blind users in Italy, that has been developed at IROE-CNR.

A major problem for blind users has recently emerged because of the transition from text-based User Interfaces to GUIs, which is mainly due to the inability of traditional text-based screen readers to
provide access to this new generation of User-Computer Interfaces. The employment of raster displays is restricting conventional methods based on translation of the video buffer since graphical screen contents are internally represented as an array of pixels. Consequently, blind users are faced with the problem of not being supported by the existing assistive technology to enable them to catch up with the new technological trends. Moreover, a general methodology for a-posteriori technical adaptations is frequently inappropriate and is not considered the most promising approach to solving this problem. Nevertheless, solutions have appeared which adopt a higher level type of filtering aiming to reproduce, where possible, an internally structured image of the visual display (off screen model). An example of commercially available solution in this field is OUTSPOKEN™ (reviewed in [GUIB92]), developed by Berkeley Systems Inc., which attempts to make Macintosh software accessible to blind users through the use of synthetic speech. OUTSPOKEN™ mainly provides access to textual information within the graphical environment but does not enable access to the various interaction objects. However, SYSTEM 3™ (reviewed in [GUIB92]), a prototype implemented at the University of Wisconsin, Trace Research & Development Centre, introduces a more satisfactory approach to cope with graphical objects. Similar efforts have been made in order to enable access to MS-WINDOWS™. SLIMWARE WINDOWS BRIDGE™, by Syntha-Voice Computers Inc., is a tool that aims to enable blind users to access MS-WINDOWS™ 3.1. A significant number of interaction objects can be recognised and textual interpretations are provided through speech, while interaction is still based on the Mouse.

Related research has been also devoted to graphical applications based on the X WINDOW SYSTEM. In [MYNA92] a method for mapping the user interface of an X-client to an appropriate auditory representation is described. This approach takes advantage of the Editres protocol of X WINDOW SYSTEM, Release 5, which defines a method for querying an X-client about information related to widgets (i.e. hierarchy, resources, etc). Also, some specific widgets that are considered meaningless for a blind user are excluded from the auditory reproduction. However, this method merely concerns access to interaction objects; for instance, it is not possible to provide access to graphical information and to interpret the specific dialogue concept for a particular push button (“on the fly”), if an arbitrary pictorial representation is employed.

A significant number of commercial solutions have been developed in the past years, which enable accessibility by blind users to User Interfaces implemented for character based terminals. It is expected that in the near future graphical User Interfaces will dominate the market, considering that software firms tend to employ graphics based interaction for their products, and also that the vast majority of firms that still produce text-based interfaces plan to convert to graphical versions. Currently, the TIDE-GUIB project aims to identify and provide the technological means to ensure continued access by blind users to the same computer-based interactive applications used by sighted users. The short term goals of the TIDE-GUIB project is to concentrate on specific developments through the implementation of appropriate demonstrators enabling access to MS-WINDOWS™ (PCs) and to interactive applications build on top of the X WINDOW SYSTEM (UNIX™ based workstations), while the longer term goals concern the development of innovative User Interface software technology aiming to guarantee access to future computer-based interactive applications by blind users. In this perspective, the work that is being carried out by the TIDE-GUIB project is taking a more global view of the problem by also investigating a variety of issues related to blind user interaction in a graphical environment, particularly for blind users. For example, different input methods which can be used instead of the mouse are investigated, the problem of how blind users can locate efficiently the cursor on the screen is being studied, the issues related to combining spatially localized sounds (both speech and non-speech) and tactile information in order to present available information are examined and the design and implementation of real-world metaphors in a non-visual form and the development of an optimal method to present graphical information from within applications are addressed.
2. Modelling Framework of Investigation

Clearly, accessibility restrictions, with respect to existing interactive systems, are due to the restricted capabilities and the various technical imperfections of the underlying interface development tools. Considering that the ultimate objective of this task is to specify the appropriate technological means in order to facilitate the generation of dialogues concurrently accessible by sighted and blind users, it is clearly necessary to investigate the relevant architectural and modelling domains. In this perspective, various selected User Interface models are presented in accordance to some well known development architectural scenarios which allow a suitable mapping from the model domain to implemented interactive systems.

2.1. Models of Human-Computer Interaction

A prerequisite in any effort towards automating the development of User-Interfaces, is a careful study of the various aspects of Human-Computer Interaction. Such an examination will provide amongst other things, an informative insight, concerning the major developmental issues involved in the design and implementation of user-computer dialogues. It should be noted that the vast majority of the investigated models provide implicitly structural properties related to the organisation of the various software layers which form an interactive system.

The 3 level model

![Behavioral Model of Human-Computer Interaction](image)

Fig.2.1.1. : A Behavioral Model of Human-Computer Interaction

Multi-level approaches of modelling interaction (originally found in [FOLE84]) can be augmented with behavioral issues. One such model is presented in Figure 2.1.1. [HOPP86]. Physical interaction is performed only at the lexical level, while additional levels present behavioral, functional and
conceptual concerns involved.

- **Semantic Level**
  Concerns the semantic services provided to the user by the application system. It is the original functionality which is augmented and accessed via the user-computer interface.

- **Dialogue Level**
  It is related to the dialogue structure and its syntactic rules.

- **Lexical Level**
  Concerns the structure of I/O items and the physical exchange of information.

It should be mentioned that this model does not impose any explicit organisational aspects; it merely introduces some fundamental structural principles related to the bidirectional man-machine communication process. It is believed that this model provides a sophisticated and generally applicable conceptualisation of the dialogue process and it can be appropriately adopted by means of a principle formal modelling pattern. For instance, particular dialogue structures may correspond to appropriate abstractions of lexical primitives by hiding the various physical properties; typically, such abstract dialogue objects would be translated to suitable physical constructs at the lexical level.

**The MVC Model**

![The Model-View-Controller Model](image)

The Model-View-Controller (MVC) is an architectural model for building interactive systems that is explicitly supported in Smalltalk-80 (an outline of MVC is found in [BARK91]). The essence of MVC
is to invariably provide a clear separation between the internals of a system (functional core) and its corresponding user interface(s). In this model, an interface consists of the following three layers:

- **Model or application layer** that implements the application functionality. This component is also relevant to the internal information structures.

- **View or presentation layer** which implements the mechanisms for presenting various aspects of the application layer to the user. In graphical user interfaces, the view could provide the constructional physical mechanisms.

- **Controller or virtual terminal layer** that handles the user’s interaction with the application. This is usually (with respect to visual interfaces) a graphics library (such as GKS) that hides device dependency and presents a device independent interface to the presentation layer.

The controller accepts input from the user and invokes the appropriate function from the model to perform the task requested by the user. When the work is done, the function in the model sends messages to the view and controller. The view updates the display in response to this message, accessing the model for further information, if necessary. Thus, the model has a view and a controller, but it never directly accesses any of them. The view and controller on the other hand, access the model’s functions and data, when required.

### The Event Model

According to this model the user-computer dialogue is viewed as bidirectional communication between the user and the interactive system via event messages. Events are either generated by the user - input events - and are processed by the system or they are initiated by the system and are perceived by the user - output events - (see Figure 2.1.3.). The interactive dialogue process is realised by a sequence of input and output events. The system logic for producing the appropriate output events is internally represented by means of event handlers; an event handler specifies the appropriate reaction to particular external phenomena (events). The event model has been widely applied in various window-based graphical environments and its fundamental constructs (e.g. events and event-handlers) are usually reflected in the implementation-oriented philosophy. It is noticeable that the event model provides considerable flexibility with respect to applicable level of abstraction related to I/O events (logical or physical events). For instance, an output event vary from a single lower level graphic primitive (e.g. lines, circles, strokes, etc) to higher-level output operations such as displaying complex graphical objects.

A formal framework for describing interactive systems complying with the event model is proposed in [HILL86], [HILL87] through the introduction of Event-Response Systems (ERS). This method encapsulates constructs which, due to the fact that enable concurrent properties to be easily expressed, have proved suitable for specifying multithreaded dialogues (via event-handlers). The model is presented in accordance to a theoretical analysis by comparing, with respect to some expressive scope criteria, with other traditional formal methods such as state automata. ERS are considered very powerful in the framework of describing easily various concurrency and synchronization aspects of the Human-Computer dialogue.
Fig. 2.1.3: The Event-Based Model

Fig. 2.1.4: The Presentation-Abstraction-Control (PAC) Model
The PAC Model

The Presentation-Abstraction-Control (PAC) [COUT90] is based on the multiagent organisation model of an interactive system. According to the multiagent model, an interactive system is structured as a collection of specialized agents which produce and react to events (however, incoming events may not only be originated by the user). PAC recursively defines structuring as a hierarchy of agents. An agent defines competence at some level of abstraction. It is a three facet logical cluster which includes:

- **Presentation**, that is a perceivable behavior.
- **Abstraction**, that is the functional core which implements some internal services and defines an interface to other agents.
- **Control**, that links and Abstraction to a Presentation and maintains relationships with other agents. The C-part of an agent may communicate with corresponding C-parts of hierarchically higher or lower agents.

According to the original formalism, a Presentation and its related Abstraction never communicate directly but exchange data in their own formalism through a common Control. The interesting point with the PAC model is that I/O specific issues are integrated (in contrast with MVC) in the Presentation, and furthermore, no explicit assumption is made as to how input / output processing should be modelled. Also, the model aims to capture the basic constructional and organisational issues involved, rather than pre-supposing any explicit behavioral properties.

2.2. Mapping UI Models to Implemented Interactive Systems - UIMS Models

The key idea behind UIMS was the provision, for each application system, of an interactive front-end that would be application code independent; that is, as much unrelated to the application semantics as possible. UIMS are considered to provide a 4th generation development method for User-Interfaces. In principle, programming effort is efficiently eliminated by declarational and descriptional techniques concerning Human-Computer dialogue. In the literature there exist several operational definitions of a UIMS in terms of the supported capabilities as an integrated tool, some of which are (reviewed in [LOWG88]):

- A tool used by a User Interface Administrator to build User-Interfaces for applications, much like a Data Base Administrator uses a Data Base Management System to deal with long term storage problems for applications.
- Something is a UIMS if it has something that determines the sequence of valid input (a parser equivalent).
- A UIMS provides a way for a designer to specify the interface in a high-level language. The UIMS then translates that specification into a working interface, managing both the details of the display and its associated input and output and also the interaction with the rest of the program.
- A UIMS is a tool (or tool set) designed to encourage interdisciplinary cooperation in the rapid development, tailoring and management (control) of the interaction in an application domain across varying devices, interaction techniques and User-Interface styles. A UIMS tailors and manages (controls) user interaction in an application domain to allow for rapid and consistent development. A UIMS can be viewed as a tool for increasing programmer productivity. In this
way it is similar to fourth-generation languages, where the concentration is on specification instead of coding.

It is worth mentioning that definitions, which have been provided in different periods of time, had a different epistemic basis according to the particular trends on UIMS development research. The initial ideas can be discovered in early periods of computer technology, when Newman was discussing about a module called "Reaction Handler".

2.2.1. Traditional Seeheim Model

In Figure 2.2.1.1., two different views of the Seeheim model are presented. This model is mostly oriented towards the physical and conceptual separation of the interface specific components from the application functionality. Weak and primitive communication mechanisms with the application are suggested, and a multilevel (time inefficient) routing of application oriented information is enforced from the lexical level to the semantic level and vice versa. The left side representation of the Seeheim model consists of the following components:

- **Application interface model**
  Holds necessary information to internally interface with the application. Usually application is treated as a semantic server by means of callable routines.

- **Dialogue manager**
  Holds information about the structure of dialogue and its syntactic rules. Mediates between the presentation component and the application and controls sequencing of interaction.

- **I/O manager**
  Holds information about the structure and syntax of I/O items. It is concerned with screen allocation and physical exchange of information. Various libraries of compact interface components may be utilised.

- **Representation manager**
  Holds information about the correspondence of I/O objects to internal application objects.

From Figure 2.2.1.1. it is apparent for which level of human-computer interaction, each UIMS component is responsible. The Representation manager communicates with both the I/O manager and application, but also accepts feedback from the dialogue manager. For this reason, it is not defined formally, in which level it belongs.

At the bottom of Figure 2.2.1.1. a more compact representation of Seeheim model is presented. Here the main components are presentation, corresponding to the I/O manager of the previous model, and dialogue control, that incorporates dialogue management. The knowledge to communicate with the application, and the correspondence of application data to presentation structures, is reserved completely by the dialogue control module. As a result, the representation manager component is implicitly included within dialogue control. The last representation looks much more relative to the 3 level approach of human-computer interaction.
2.2.2. The Arch Model

The lexical level technology toolkit usually poses considerable constraints on the scope of the User-Interfaces that can be developed using a particular UIMS. Constraints may exist due style restrictions or due to limited functionality support. A similar problem is detected when dealing with issues concerning the communication with the functional domain components (application). The UIMS
software must manage the communication between these components as it was explicitly defined within the Seeheim architectural framework. Similarly, these levels form the basis of the Arch model [UIMS92] (see Figure 2.2.2.1.). However, additional intermediate levels are introduced to achieve an elastic behaviour concerning functional and presentational capabilities. A detailed description of the five components of the Arch model follows.

- **Domain Specific Component**
  Controls, manipulates and retrieves domain specific data and performs other domain related functions.

- **Interaction Toolkit Component**
  Implements the physical interaction with the end-user (via hardware and software).

- **Dialogue Component**
  Has responsibility for task-level sequencing, both for the user and for the portion of the application domain sequencing that depends upon the user; for providing multiple view consistency; and for mapping back and forth between domain-specific formalisms and User-Interface specific formalisms.

- **Presentation Component**
  - a mediation, or buffer, component between Dialogue and Interaction Toolkit Component that provides a set of toolkit-independent objects for use by the Dialogue Component (e.g., a "selector" object that can be implemented in the toolkit using either a menu or radio buttons). Decisions about representation of media objects are made in the presentation component.

- **Domain Adaptor Component**
  - mediation component between the Dialogue and the Domain-Specific Components. Domain-related tasks required for human operation of the system, but not available in the Domain-Adaptor Component triggers domain-initiated dialogue tasks, reorganises domain data (e.g. collects data items in a list), and detects and reports semantic errors.

It should be mentioned that the term object is used here in general without posing any guidelines or constraints on the actual implementation concerning communication entities being exchanged. Instead, objects are used here as an expository abstraction for describing communication mechanisms. A detailed description of the role of each type of object follows.

- **Domain Objects** are used by the Domain-Specific and Domain-Adaptor components but with a different purpose concerning their instantiation. In Domain-Specific-Component, Domain-Objects encapsulate domain data and underlying provided functionality (methods) which can be applied on domain data. Domain-Adaptor-Component deals with Domain Objects, but provides an adaptation mechanism by the implementation of specific services, according to required formatting and management of Domain-Objects depending on presentation and interaction requirements.

- **Presentation Objects** provide abstraction of interaction objects in an implementation independent form in terms of collections of virtual interactive entities. Specific technology dependent implementations constitute particular interactive instances. Presentation objects incorporate data to be accessed by the user and also a well defined, implementation independent, event driven user access method.

- **Interaction Objects** are the specific implementations of Presentation Objects. They are supplied by the lexical level toolkit which has been selected for a particular implementation of the user-computer dialogue. The presentation method and the actual (device oriented) event generation mechanism are provided at this level.
2.3. Discussion

Existing UIMS models deal with the functional structure of an interactive system with respect to a higher-level implementation methodology. The traditional Seeheim model which manifests the fundamental principles in this domain, has been subject to considerable critique in the past; however, it is believed that various diverse misinterpretations of the original model has driven to a number of incorrect assumptions and unrealistic conclusions concerning matters of abstractability, universality and practical applicability. It is argued that the basic model could be also faced by means of a meta-model in which the various levels may be realised by means of appropriate sub-structuring in the specific reference models. For instance, the lexical layer could be viewed as the meta-level of software structuring corresponding to software components that deal with particular physical aspects. In this perspective it is considered that the Arch model may be normally considered as a particular instance of the basic Seeheim model (in literature the Arch model is considered to follow the Slinky metamodel [UIMS92]). In the proposed approach, the dialogue level must be completely independent of any lexical concerns by shifting physically relative issues to suitable lower levels.

Fig.2.2.2.1. : The Arch UIMS Model

Currently, various formal techniques for the description of user-computer dialogue are available. Such methods have differing expressive properties and abilities and typically reflect the adopted underlying dialogue model. In the perspective of designing a suitable dialogue-specification formalism for the proposed framework, a careful investigation of such existing methodologies has been conducted with the aim of identifying particular technical aspects and language characteristics which could be effectively adopted for the design of the proposed interface specification technique. The analytic study in this specific field has led to an appropriate synthesis by means of an informal classification scheme which is presented below.

3.1. Quantitative Classification

A number of key issues are addressed here, concerning the various formal specification methods which are available (to dialogue designers), in terms of their expressiveness and appropriateness with respect to modelling efficiently different aspects of user-computer interaction. Such techniques, vary significantly in terms of the notational complexity, applicational convenience and descriptive suitability. Such an examination is necessary, in order to identify the various required language characteristics that are suitable for describing Dual Interfaces. For instance, notational facilities which incorporate presentation independent constructs have been duly taken into consideration during the design of the proposed formal framework.

Task Oriented

Task oriented methods are based on the behavioral aspects of the interactive system. They adopt a modelling of dialogue based on tasks the user has to accomplish and they aim to describe formally the natural behavioral issues involved during the interaction. Tasks may consist of sub-tasks and several temporal constraints may describe accessibility restrictions or impose synchronisation of actions. Asynchronous behaviour and concurrent dialogues can be also formally supported in some languages. Informally, a task is a collection of activities in a domain with a common goal. Task oriented description does not imply task awareness. Task aware-interfaces represent in machine readable form the knowledge about [PAVL90]:

- what the task is
- how it is presented in the interface
- how it is decomposed into actions
- how these actions correspond to the applications’ functionality.

There is an initial state which is either explicitly (physically) or conceptually represented. There is a target goal that has to be achieved and also a set of available operators (methods) which can cause a specific state transition. The completion of a task is defined by the transformation from initial state to goal state via mental and physical operations. The main attributes of a task are: purpose, goal, stopping criterion and plan. TAG [REIS81] and UAN [HART90] are task-oriented dialogue description techniques.
Event Oriented

Event oriented notations describe formally an event based interactive system which receives various types of events from multiple sources (such as application originated or user generated), and performs a predefined associated set of well defined actions. Such functioning of accepting events and executing appropriate actions is a repeating process which continues until particular terminating conditions are fulfilled. The notion of event handlers is implicitly or explicitly introduced in such notations. Event handlers can be generally considered as rules to be executed when specific types of event based conditions are fulfilled. The event model features an implicit I/O event ordering, where the specification defines sets of events without mentioning a specific ordering. Ordering restrictions are implicit in the UIMS that interprets the specification. ERL [HILL86], [HILL87] is an event-based notation.

State Oriented

State oriented User-Interfaces provide a means of describing the interaction dialogue in terms of specific dialogue station points. States can be considered as a method of temporal labelling where the dialogue status is well specified by the various active states. During the interaction, entering (activating) or leaving (deactivating) a particular state is determined by various conditions the satisfaction of which may trigger specific state transitions. The most fundamental state oriented natational technique are state transition networks. They have been in use for a long time in computer science because of their relation (and party equivalence) to finite automata. Various alternative schemes, successors of traditional state transition networks, such as ATN (augmented transition networks), RTN (recursive transition networks), Petri-Nets and State-Charts, had been progressively adopted which effectively contributed to their original descriptive power. However, the lack of more natural easily manipulated description methods and the fact that notational complexity was tremendously increased on large interactive systems, introduced serious applicability problems and gradually turned them to less preferable dialogue portraits. StateCharts [WELL89] and STN [JACO88] are state-based formal methods.

Object Oriented

In currently available graphical environments the lexical level of user-computer interaction is usually built upon toolkits of interaction objects. Such dialogue entities, the development of which has been based on the notion of providing natural interactive metaphors, have contributed revolutionary in the design and implementation philosophy of graphical User-Interfaces of the past half-decade. There are a lot of dialogue description methods that are centred around interaction objects. Various characteristics (behavioral or physical) of objects are defined by the dialogue designer and appropriate linkage with application domain is explicitly specified. The nature and extensibility of objects vary significantly. It is common in various UIMS to supply fixed libraries depending on a specific lexical level toolkit. More flexible UIMS support the integration of theoretically arbitrary toolkits by the provision of an efficient interface mechanism [5]. In other UIMS a set of standard generic interaction methods, called interactors, is provided in accordance to a suitable mechanism for combining them to composite interactive objects [4].

In various systems, developers build a major portion of the User-Interface by declaring objects and assigning values to object attributes. This enables developers to focus on defining the contents of a User-Interface without concern on the procedural aspects of its implementation. Accordingly, a declarative model is easily and efficiently learned by non-programmers. However, declarative methods can become insufficient when facing the specification of a User-Interface that may change dynamically on the basis of complex conditions. In order to overcome this problem it is possible to introduce procedural constructs that may be combined appropriately with interaction objects [MANH89]. LUIS
[MANH89], GARNET [MYER90], NeXT Interface Builder [THAK90] and SERPENT [BASS90] are UIMS that are based on interactive entities for the construction of user-computer dialogue.

Semantic Oriented

A common attribute of the previously mentioned dialogue description methods is that the dialogue designer has to explicitly specify the various lexical and syntactic details of the interaction. Semantic oriented descriptions are accompanied by special tools that alleviate some of these problems by automating parts of the designer’s work. A detailed description of the application domain supplied semantic services and of their various functional properties is considered adequate for the automatic production of the dialogue. Moreover, a user model can seriously influence the selection of the most appropriate interactive characteristics of the produced dialogue component. Mickey [OLSE89] and UofA* [SING89] are semantic oriented UIMS.

Definition by Demonstration (Show-by-Example)

The essence of such specification techniques is the explicit derivation of user-computer dialogue from informal demonstrational specifications. The interface designer specifies interactively some of the interactive behavioral properties or presentation sketches from which a possible dialogue is automatically generated (it is not necessary to define in complete detail every interactive attribute). Usually special purpose interface editors are provided where the user can construct visual display instances with common (interactive) graphic entities. The designer is interacting directly with graphical objects from a special natural visual language rather than having to refer indirectly to verbal symbols. PERIDOT [MYER88] supports definition by demonstration.

Display / Presentation Oriented

In this specification category, the physical appearance (lexical level) of the user-computer interface is the main design perspective. Efficient interactive (WYSIWYG) screen design tools are usually provided which allow the designer to place objects on the screen and build the constructional view of the dialogue. Events may also be introduced within objects as a means of calling specific semantic functions or triggering predefined sequences of output operations. The dialogue architect may select the type of an object, when it should be displayed or destroyed, accessibility constraints, physical appearance, etc. The connection with the application domain is achieved through special notational conventions. MENULAY [BUXT83] is a display oriented UIMS.

Rule Based and Constraint Oriented

Such techniques allow declarational specification of the user-computer dialogue; for instance, it is required only the definition of the various properties and particular governing the interaction in accordance to actions which would be triggered when specific conditions may be satisfied. Actions and conditions may be presentation oriented, controlling and settings of flags, or calling of application domain oriented services. Constraints have been proved to be a powerful method for defining behavioral properties declaratively. A well known method of describing conditions and associated actions are Production Systems, which model constraints in condition-action pairs. Another method is the introduction of special constraint objects that take care of the satisfaction of specific conditions and may trigger appropriate predefined actions. PPS [OLSE90] is based on propositional logic, PERIDOT [MYER88] is based on geometric constraints, and THINGLAB [BORN81] which was one of the earliest constraint-based systems fall in this category.
3.2. Qualitative Classification

A qualitative classification aims to categorise the various specification methods according to the major aspects of user-computer dialogue that are explicitly covered. A specification method may be oriented towards the visual structure of the display (in terms of its contents and layout spatial relations), the behavioral aspects of human-computer interaction or the application domain supplied functionality.

Morphological Methods

Some specification methods are oriented towards the physical structure and the individual entities that comprise the display during interaction. Notational conventions for the definition of display oriented topological/spatial constraints and physical interrelationships are usually supplied. Low level graphic primitives can be combined for the structural description of display appearance. Moreover, interaction object based specifications facilitate the effective layout design.

In constructional methods, the dynamic features of user-computer dialogue may be explicitly specified through: definition of sequences of various output operations, association of graphical entities with application oriented functions or establishment of conditional actions. Object oriented, display oriented and show-by-example techniques lay within this constructional domain. Clearly, this category necessarily incorporates presentation oriented constructs with their various formal dialogue building-blocks. Consequently, similar methods are considered inappropriate when dealing with the description of Dual Interfaces.

Behavioral Methods

Behavioral description methods are directed towards the specification of the nature of actions either from a user oriented perspective (what the user has to accomplish and how) or in terms of the system (how the system has to react and when). Also, description of temporal properties (e.g. sequencing, synchronization, etc) is supported in certain systems. The provided notational formalism is based on these issues, and some times interactive (display oriented) definitions are possible. It is significantly important to facilitate, through special purpose tools, formal representation of knowledge related to the work the user has to accomplish (in a specific domain) because it allows the construction of adaptive / intelligent interfaces. Task oriented descriptions usually incorporate such features. Sometimes only machine’s behaviour (rules of reaction) is explicitly modelled; mainly, the behavioral characteristics concerning the actions to be carried out by the interactive system on specific user input are formally expressed. Rule based and constraint oriented techniques are appropriate for supporting such behavioral properties.

Functional Methods

Semantic methods are based on the underlying functionality of the application domain. The specification method aims to capture the major aspects of domain oriented services that need to be interfaced (accessed) appropriately to the user. The behavioral protocol and presentation issues are usually transparent in the formal specification method and are implied automatically from the high level application description. Even though the previous characteristic relieves the designer from the burden of explicitly introducing behavioral and presentational attributes, it poses serious restrictions on the final decisions (e.g. customization) concerning such issues. In order to overcome this problem, various additional formal methods in order to specify particular behavioral characteristics or to set stylistic rules, have been additionally introduced in terms of formal hints to the interface generation
Hybrid Methods

Hybrid methods combine characteristics of the previous methods. For example, a functional method may allow fine-tuning (editing) of the behavioral properties for automatically generated dialogues. Also, various existing constructional methods also provide powerful notational facilities for the specification of behavioral characteristics. Intuitively, in this context it may be desirable to couple visual and non-visual constructional methods and provide a unified presentation independent notation.

3.3. Further Discussion

Some specific topics that are addressed separately are related to user-modelling facilities and efficient on-line help handling. In this context, a lot of partial solutions and methodologies exist, however, it is argued that none of these is provided in terms of an integrated dialogue specification scenario.

3.3.1. User Modelling Facilities

Currently, there is no standard representation of what actually constitutes a user model. Moreover, the existing terminology is heavily confused by ambiguity about who constructs the model (subject) and what is being modelled (object). User models can be characterised about user’s view and understanding of the system (mental models) or even the designer’s idea of how users realise the system (conceptual models). Some of the existing models (which lie between these two categories) in the current human-computer interface literature are the following (reviewed in [SUTC88]):
Fig. 3.3.1.1: The 3-Dimensional User-Computer Knowledge

- **Theoretical cognitive models** constructed by psychologists in order to understand human mental processes. Information processing models, fall into this category.

- **Models of User Knowledge.** These models are inspired by CBT (Computer Based Training) interests and adaptive interfaces. The model aims to capture the various knowledge categories in a domain and particular inter-relationships between the categories. Reference models may then be constructed concerning each user’s knowledge which may be subsequently used in order to automatically induce how users learn, by traversing the knowledge network. In adaptive interfaces the model attempts to describe the user’s knowledge in terms of plans and procedures. These models are embedded in software.

- **Models of User Characteristics.** These models attempt to classify users in broad terms of skill and ability. They are also called user profiles.

  - **User Task Models.** The user’s concept of how a task is constructed in terms of its functions and operational sequence.

  - **User Views.** The user’s models of the system structure which may be expressed either in terms of visual metaphors (for example, an office and its components) or in a verbal classification of system components. They are also called the user-system image.

### 3.3.2. Efficient On-line Help Handling

Even on excellent User Interface designs it is not possible to ensure that the user will always understand and realise correctly every interactive phenomenon. Moreover, it is utopian to consider that a careful design can exclude, during interaction, planning support deficiencies, failure related to target
goal satisfaction or even user disorientation. Usually, the descriptive capability of available concept representation methods via graphic oriented symbolisms (metaphors), is determined by complex cognitive and perceptual rules. For an instance, icons must look like what they mean; they have to be representational and easy to understand. Anyway, it is obvious that our understanding of events and ability to predict outcomes is always imperfect. For this reason, help provides an insurance policy against less-than-perfect design. Some of the questions a user might have while using a program (original description found in [KEAR88], italics indicate additional comments) are listed below:

- What can I do next ? (alternatives/ordering)
- How should I respond ? (legitimate input)
- What does this mean ? (descriptive adequacy)
- Where am I ? (orientation)
- How can I do this ? (strategy/planning/goal satisfaction)
- What happened ? (feedback adequacy)
- How can I change this ? (active engagement/strategy/planning)
- Why didn’t that work ? (reliability/feedback adequacy)

The above is only a representative set of questions. Clearly, some of them are due to lack of sophisticated dialogue design and not because of the limited power of the dialogue development tools. However, an efficient on-line help facility should be able to prevent (and detect) at an early stage such design problems by providing automatically sufficient and appropriate user support.

3.3.3. Critique

Task-oriented representations have been realised by existing systems in a restricted abstract form; when the decomposition of user-activities into tasks involves primitive tasks, specific physical issues are implicitly or explicitly engaged. Moreover, relevant methodologies are usually confusing with respect to the semantic relation between the task-model and the user’s mental system model; typically, the task-description concerns a specific user-group and consequently it becomes practically inapplicable for other user-groups. One might argue that such a conclusion is contradictory due to the fact that task-oriented knowledge is mostly related to the organisation of user activities (without, in principle, representation of physical characteristics) in order to accomplish a specific goal and intuitively comes in accordance to the fundamental requirement of supporting formal constructs free of lexical attributes. However, in this case the problem is due to the incorrect assumption that both user-groups could normally share a common task-model; it is believed that existing notations are too restrictive and syntactically inflexible and would necessarily require that a blind user should carry out the same interactive steps in order to reach a common goal.

Event-based languages have significantly powerful expressive properties and are considered very flexible dialogue description tools. Complex input sequences and concurrency aspects can be efficiently modelled. However, these methods are very weak with respect to describing output structure; an explicit separation of input / output events is required and moreover it is very difficult to provide display organization through output events due to the fact that events do not have constructional properties and are also structurally primitive (e.g. cannot built hierarchical aggregate blocks) which practically does not allow reusability of common synthetic output patterns. Considering that in order to support the higher-level development of direct manipulation oriented interaction it is crucial to allow re-using notationally output as input, it becomes evident that it is required to augment the basic event-model with constructional characteristics. Nevertheless, a worth-mentioning advantage is the provided freedom to define logical events which may physically mapped to frequently repeated sub-dialogues; subsequently, the unification of composite and radically different lower-level activities under the same abstract event classes is enabled for both user categories.

In our domain, state-based descriptions are considered inappropriate because of the primitiveness of the dialogue description constructs and the proved inappropriateness to capture higher-level
presentation independent properties of the user-computer dialogue and consequently support syntactically free formal expressions. However, the object-oriented paradigm has been given considerable attention because it formally combines various properties of the previously mentioned dialogue description methods. For instance, objects are suitable abstractions of I/O events and indeed have powerful constructional abilities. Also they are modal free; the user may asynchronously interact with a particular object and every object is directly accessible assuming it has been displayed. The introduction of lexical attributes for interaction objects within the dialogue specification method restricts applicability for non-visual dialogues however, proper conceptualization of such interactive entities would enable uniform notational manipulation irrespective of the instantiation of such abstract dialogue constructs to environment (visual or non-visual lexical technology) specific physical entities.

Constraint-oriented and rule based methods are characterised with their abilities to declaratively express complex relations between the various User Interface components. Update propagation and multiple-views of information may be usually efficiently handled by the designer without any concern on management aspects and explicit procedural issues. For example, in case that shared internally information should be represented to visual and non-visual form an appropriate rule may be associated which would constrain that data modifications (either originated by the user or caused by the application) should be automatically reflected to both physical environments.
4. The Proposed UIMS-Based Architectural Framework

The effort of designing the architecture of an appropriate framework, which meets the various identified functional requirements, has concluded with the finding that at present, the software technology for UIMS is the most suitable epistemic basis. In this context, the various issues related to dialogue specification methodologies and run-time management scenarios have been exhaustively addressed in the perspective of User Interfaces concurrently accessible by sighted and blind users. Also, even though the proposed approach has been designed in order to achieve integration of sighted and blind users, considerable attention has been paid so that easy and efficient development of conventional dedicated visual or non-visual dialogues is also supported by the original methodology.

4.1. Terminology

One of the primary goals of this research effort is to support the reproduction of dialogue concurrently in visual and non-visual form; the User Interface is required to be collaborative to both sighted and blind users, running on a single workstation where additional appropriate peripheral devices need to be installed in order to provide non-visual interaction. According to the proposed framework, a User Interface meeting these basic criteria is called a **Dual User Interface** (see Figure 4.1.1.). However, in order to ensure that the concept of Dual User Interfaces is, indeed, meaningful and has a considerable practical value, some additional properties are also defined.

![Diagram of Conceptual Structure of a Dual Interface](Image)

Fig.4.1.1. : Conceptual Structure of a Dual Interface
Following the introduction of the concept of Dual interfaces, the notion of a Dual Lexical Level is put forward, which reflects the necessity for supplying an alternative non-visual physical level as a counterpart to the visual one. Additionally, the notion of a Dual Display Environment (implicitly included in the Dual lexical level) provides a comprehensive framework for visual and non-visual output. A non-visual display is likely to differ considerably in terms of its physical and topological properties from conventional visual displays. Various special output devices may be utilised for the construction of the alternative display framework, some of which could be employed in the visual environment as well. In this report, a suitable methodological framework is proposed, which enables a higher-level (without programming effort) development of Dual User Interfaces, in the context of User Interface Management Systems (UIMS) software technology.

4.2. Principal Functional Requirements

Some of the following requirements are related to the necessary notational scope of the dialogue specification tools, while others concern specific run-time management issues. However, all of them are considered equally important for supporting the effective and efficient construction of Dual User Interfaces.

Eliminate restrictions concerning visual or non-visual dialogue design, due to integration in a common interface

It is evident that the interface designer will have to draw a dialogue portrait following a methodology that will take into consideration, and finally reflect, the required duality of the eventual interactive system. Hence, the supplied specification methods must allow flexible definition of the design concepts, which will properly meet the specific needs of both user groups, through the provision of an appropriate dialogue structuring methodology that should allow bridging of the various visual and non-visual interaction pictures in a unified formal structure.

Support of information multimodality (multiple accessible views)

The essence of information multimodality is to provide, simultaneously, multiple points of view for identical information, enabling further manipulation by the user via various alternative channels. Additionally, interactive methods for supporting user input are considered critical. Automatic change propagation, concerning externally or internally caused updates of information, is indispensable for supporting this feature. Internally shared information, is likely to be mapped to different output structures on both (visual and non-visual) display environments. Moreover, considerable flexibility must be provided to the designer for the organisation of presentation, and for the definition of interactive access methods; in this context, the common denominator between interaction and presentation methods, regarding visual and non-visual environments, are the underlying internal pieces of data. Information multimodality, is a commonly applied technique for highly interactive direct manipulation interfaces; the multidimensional nature of direct manipulation methods should be noted in the perspective of such multimodal representation of information.

Allow utilisation of a variety of (visual / non-visual) lexical technologies

Considering that the proposed approach needs to provide adequate flexibility (to the designer) for the definition of the lexical pragmatics of dialogue (e.g. the lexical level properties of Human-Computer dialogue), it is evidently important to allow utilisation of available lexical technologies (i.e. conventional toolkits). However, at present, there is no implemented framework for handling the
lexical level of interaction in a non-visual form; this implies that the provision of an appropriate mechanism, in order to incorporate various lexical technologies, is not just an additional functional flexibility, but a critical and fundamental architectural prerequisite.

Preference on syntax free and flexible dialogue specification techniques

The idea behind syntactically flexible techniques, is to enable a considerable amount of interactive freedom (to the user), that would otherwise be feasible only through the provision of multiple specifications of the involved user tasks, following a typical syntax oriented formalism. In principle, adoption of similar techniques, could facilitate unification of dialogue specifications; even from a common dialogue description, no identical sequencing of user actions is explicitly required. Intuitively, imposing on two user groups with significantly differing abilities and requirements to follow a common interactive behaviour, would probably cause serious problems to either or both user groups.

Enable the integration of various (special purpose) peripheral I/O devices

Currently, visual toolkits are inefficient in handling the installation of additional devices for subsequent utilisation at the physical level of interaction with the user. In order to support the provision of a non-visual lexical environment, various special I/O devices are required (i.e. 3D pointing devices, speech synthesizers, tactile modules, loudspeakers of stereo sounds, etc). However, in case that such devices can not be handled directly through the non-visual lexical technology, an appropriate mechanism is
required, which would allow their integration and enable the designer to formally specify ways of appropriate exploitation during user-computer dialogue (e.g. extension of input techniques must be allowed through translation of input functions to specific new event types, when extra devices must be installed).

**Fig. 4.2.2. : An Example Demonstrating Transparent User Augmentation**

**Enable support for transparent user augmentation**

The designer should be allowed to provide additional interactive features (quantitatively) for a particular user group, in order to augment the dialogue efficiency, without affecting the principle of qualitative equivalence on the formally specified interactive methods; either user group may equally intervene on represented concepts, but through differentiated interactive facilities. The required qualitative similarity between visual and non-visual dialogues, as defined in this context, is not decided by means of cognitive criteria (obviously, transparent user augmentation is heavily related to cognitive and perceptual augmentation), but in the perspective of the general (user) ability to manipulate the underlying application oriented represented concepts. For instance, an appropriate interaction object, serving the purpose of a command-line for an operating system, should be provided for both user groups simultaneously; furthermore, the designer may decide to supply transparently (concerning the blind user) additional visual interactive facilities such as a graphical representation of the hierarchical (tree) structure of the file system in accordance to an indication of the particular current and home
directories, a variety of command buttons corresponding to frequently used parameterless commands and a scrollbar for controlling the viewing of these buttons; both user groups share equivalent facilities in terms of the underlying functionality, accessible though by different interactive channels.
4.3. Conceptual View of the Analytic Design

In this context, the various key issues involved within the conceptual design of the proposed framework are addressed. Also, an informal detailed description of the designed architecture is presented; this manifests the various fundamental constructs in accordance to the specific theoretical and practical extensions, which are suggested with respect to traditional interface development systems.

4.3.1. Multilevel Structuring of the Dialogue Specification Scenario

Our investigation of the various syntactic and mostly lexical properties of Dual dialogue, has lead to the following schema of generalization, that can be also applied to conventional visual interfaces, which defines some basic dialogue categories:

- Dialogue modelled sufficiently through interaction objects, where each interaction object involved may either be display oriented (e.g. the general concept of interaction object is tightly coupled with the display environment properties; only single reproduction can be achieved) or display independent (e.g. the concept of interaction object is independent of the display properties, which enables reproduction of the same object in the alternative display environment; Dual reproduction).
- Dialogue not modelled through interaction objects (but may involve interaction objects); lower level or device oriented interaction.

It should be noted that the interactive scope of interaction objects, is a dimension of the toolkit; for instance, graphic editing may be either modelled through interaction objects or via application of lower level lexical constructs. Due to the fact that lower level interaction is tightly coupled with the display environment, it is considered meaningless for Dual reproduction. However, the principle of Dual reproduction at this level, is reduced to a guarantee of consistent and immediate propagation of the updates, concerning affected semantic concepts (e.g. shared displayed internal information), to the alternative display environment.

For example, consider a conventional programming environment, for an object oriented language, where the programmer needs to define the class hierarchies. In our approach, the class hierarchy organization constitutes shared information for both users, which could be represented in each display environment with significantly heterogenous presentation forms. In the visual environment, graphical techniques and visual symbolisms might be applied, while in the non-visual environment, conventional textual input and tactile output might be supported; automatic propagation of updates, concerning the semantics of the class hierarchy, must be distributed to both environments.

The criteria which identify the frontiers of lower level interaction are also dependent on the scope of interactive facilities supported by the lexical technology and provided through interaction objects. More specifically, considering conventional visual point & dragging of interaction objects, the toolkit may take care of dialogue management, by controlling devices, providing feedback (usually a dashed rectangle), modifying position of objects and refreshing, when necessary, affected objects. Such dialogues, involve invariably a pointing device (such as a mouse), and require continuous control over that device. In case that this lower level control is delegated to the toolkit, the User Interface designer has to deal only with higher level concepts (e.g. semantic feedback, semantic actions, syntactic constraints, etc). Otherwise, an explicit description of the dialogue is required, involving device representation (i.e. through event based constructs) and low level lexical concepts; such dialogue representation methods, must satisfy the specific needs of lower level dialogue.
4.3.1.1. Dialogue Description and Notational Constructive Tools

Recalling that interaction objects form the basic interactive mechanism for Dual interfaces, there are some primary requirements driving the design of an appropriate functional model of interaction objects, which will be employed in the proposed architecture. Even though these requirements emerged in the proposed framework of Dual interfaces, they can have a considerable positive impact on visual interfaces as well. More specifically:

- It must be possible to provide interaction objects with various lexical implementations; an interaction object may have numerous visual and/or non-visual lexical representations, corresponding to each particular lexical technology being utilised for each display environment. We call this property of interaction objects *Lexical Polymorphism*.

- Similarly, an interaction object can have various syntactic representations; even with the same lexical image, an interaction object may be provided with different behavioral attitudes. Similarly, we call this property *Behavioral Polymorphism*.

Lexical and behavioral polymorphism are some of the desired fundamental properties that must be explicitly supported in our model. Moreover, in order to provide the methodology of deciding whether it is appropriate to provide Dual reproduction for particular interaction objects, there is a necessity for defining:

- An appropriate schema of abstraction for interaction objects (see Figure 4.3.1.1.1.) that indicates clearly at which levels polymorphism is applicable.

- Selection criteria for identifying which objects are appropriate for Dual reproduction. Following the terminology of Figure 4.3.1.1.1., in case that the represented concept deals with presentation specific issues, it is considered meaningless to support Dual reproduction (e.g., a visual physical interaction object that serves specific layout management purposes, is utilised for decoration, or supports special visual effects and provides visual cues).

In accordance to the schema introduced in Figure 4.3.1.1.1., the proposed model supports a collection of abstract (Virtual) interaction objects. Elimination of the necessity for any explicit reference on syntactically or lexically oriented concepts is achieved by shifting relative issues to a lower level of specification, where through the provision of an appropriate instantiation mechanism, an interaction object may be associated with various behavioral and lexical images. Following this approach, a higher level of conceptual abstraction is introduced, in comparison to various existing methods which adopt behavioral abstraction [DESO89], [ZAND89].
An analysis of the various aspects of user input into basic interaction tasks is introduced in [FOLE74] and updated in [FOLE84]. However, the latter model is considered to be quite restricting and inadequate for highly interactive User Interfaces. For instance, structural issues are not explicitly addressed and also some of the proposed constructs are likely to be inappropriate when dealing with non-visual interaction (i.e. 2D path, position, orientation). According to our proposed model, the informal dialogue categories identified earlier are refined and adapted as follows:
Dialogue based on virtual interaction objects (Higher Level Dialogue), where any object may either be a Dual virtual interaction object which can be provided simultaneously in visual and non-visual form, or a conventional virtual interaction object, which can be reproduced only for a single display environment.

Dialogue not modelled through virtual interaction objects (Lower Level Dialogue), where an appropriate methodology must be applied in order to provide formal representation of such physical aspects of dialogue.

One significant advantage of the adopted approach is the independence of the higher level dialogue specification from the underlying syntactic and lexical structure of the virtual interaction objects involved. This property allows:

- Modular integration of the appropriate visual and/or non-visual implementations for Dual or conventional virtual interaction objects. It is expected that the syntactic and lexical attributes of visual instances will be totally different from the corresponding non-visual instances (heterogeneous implementation); however, the original dialogue description involving virtual interaction objects is not affected.

- Further modification/customization of the syntactic and lexical properties of the various physical instances (one allowed for each lexical technology being utilised in each lexical environment) of virtual interaction objects, without dictating intervention on the original dialogue description.

Fig. 4.3.1.1.2. : Structuring Formal Dialogue Specification into Higher and Lower Levels
Aiming to maximize the behavioral flexibility and applicability of the available set of virtual interaction objects, the following additional issues, related to the facilities supported by the specification methodology, have been addressed:

a) An appropriate technique enabling expansion of the available set of virtual interaction objects is required. For this purpose, a formal method allowing the introduction of additional virtual interaction objects has been constructed; the designer will have to specify, through the instantiation mechanism, an appropriate mapping of newly introduced virtual objects to toolkit specific (physical interaction objects) instances (see Figure 4.3.1.1.2.). We call this primitive specification mechanism Virtual Object Genesis.

b) In accordance with the Virtual Object Genesis, an appropriate composition mechanism is required that would enable handling of multiple interaction objects as a common entity; the designer has to further specify a physical grouping technique, based on object hierarchies, to be applied upon instantiation of such synthetic virtual objects at the lexical level. For Dual virtual interaction objects, this must be provided for both lexical environments. Similarly, we call this method Virtual Object Composition.

It should be noted that the above specification schema does not involve any presentation oriented issues. Also, genesis and composition, are only applied in the virtual world; the method does not deal with the problem of introducing additional interaction objects at the physical level of interaction. It is assumed that the various lexical technologies provide the physical interactive objects, while freedom is provided to the designer to define the mapping from conceptual to lexical structures. This type of syntactic conceptualisation of the higher level dialogue is one of the fundamental architectural issues for Dual User Interfaces, where presentational concerns are appropriately shifted to a lower level of specification. In Figure 4.3.1.1.2., the idea of organising dialogue specification at higher and lower level layers is illustrated. Using the specification technique for lower level dialogue, the designer has full notational access to the interaction objects involved on the definition of higher level dialogue; this transparent to the designer inheritance effect, is easily achieved via the instantiation mechanism.

Formal description of the higher level dialogue is possible by means of organising Dialogue Agents - conceptual processes controlling multiple virtual interaction objects. The nature of the supported specification method has notational similarities to SERPENT's View Controllers [BASS90], while the underlying implementation oriented model is based on an adaptation of the PAC (Presentation-Abstraction-Control) architecture [COUT90]. Roughly speaking, a Dialogue Agent is an augmented PAC agent having the ability to handle a Dual Presentation component. Also, instead of providing a presentation specific module which incorporates knowledge specific to the target physical platform (e.g. presentation dependent issues), a Dialogue Agent sends output requests originated from the higher-level to a Representation Agent which in turn communicates with the lower level via a standard protocol independent to the properties of the target environment. At the lower level, an Instantiation Agent dispatches higher-level requests to the corresponding lexical jobs. It should be noted that all output specific requests posed from the higher level may only concern virtual interaction objects.

A common scheme for the definition of constraints and dependencies is supported; interaction objects, dialogue variables and application items are viewed as objects in the specification method. The designer is allowed to define for any particular object, dependency on a list of arbitrary object-based expressions in accordance to condition-based selection criteria, where dependencies are treated as unidirectional constraints. At run-time, the applied constraint satisfaction algorithm, uses the object dependency-graph, that has been generated upon compilation, in order to identify propagation of effects, each time an object is being modified. A similar approach based on Active Values is found in [SZEK88].

Concerning the definition of lower level dialogue, in the area of visual User Interfaces, a lot of successful work has been carried out, sometimes covering both higher and lower level aspects of
dialogue definition (i.e. event-based models [HILL86], [HILL87], [CARL90]). A relative, but more syntax oriented methodology is found in [REIS81]. General issues concerning functional, behavioral and structural analysis of Human-Computer interaction, elaborating mostly on lower level aspects and automatic generation of User Interface code, are partially addressed in [LIU90]. However, some worth mentioning existing methods which have contributed significantly in the domain of visual interface technology, are considered inappropriate for non-visual dialogues, due to the involvement of visually oriented concepts in their fundamental constructs [MYER90], [HART90], even though in our approach they could provide an efficient modelling for the various lower level aspects of visual dialogues. State-based modelling does not impose any predefined interface constructs, and is suitable for providing executable specifications [JACO88]; however, it is difficult to support concurrency and, moreover, notational complexity is tremendously increased for large interactive systems. Refinements on the basic model have been introduced, aiming to enhance usability of such specification techniques [WELL90]. Some of the various properties and principles of such methods, which are meaningful in the perspective of describing non-visual lower level interaction, may be taken into account for the development of the lower-level dialogue description method. In particular, the following criteria and requirements should be taken into consideration:

- A common specification method must be applied for both visual and non-visual lower level dialogues; consequently, the methodology should not impose a-priori any display specific concepts.
- At the physical level, appropriate support for concurrent and synchronous / asynchronous input is required.
- Facilities for the specification of the instantiation logic from virtual to physical interaction objects are required.
- The method must allow specifications involving the physical instances of the various virtual interaction objects defined at the higher level dialogue.
- Facilities for the introduction and handling of additional peripheral I/O devices are required.
- Provision of appropriate notational conventions facilitating flexible reusability of specification items.
- Description of the lower level representation methods must be efficiently supported.

4.3.1.2. Modelling Morphological Representation of Application Oriented Information

Considering that one of the major principles of Dual Interfaces is to always ensure that identical application oriented information is represented in both display environments, it is apparent that automatic update propagation becomes one of the most important architectural concerns. The underlying management & communication mechanism should allow detection and distribution of effects consistently and efficiently. In [TREF90] a Representation Manager module is responsible for mapping internal entities to external structures and maintaining consistency. In [SIBE88], representation objects (R-objects), interaction objects (I-objects) and application objects (A-objects) constitute the fundamental dialogue management constructs; this approach aims to overcome the problems due to the communication overhead in traditional multilayer approaches, such as Seeheim. Some higher level similarities with these approaches are evident in our proposed scheme below.

In Figure 4.3.1.2.1., a general (conceptual, not implementational) structural view of the adopted method is outlined. In this model, the fundamental presentational duality of application oriented information is explicitly posed. For simplicity, the model provides a synthesized view of two implementationally different, but architecturally similar strategies. These strategies correspond to the separation of the various representation techniques in the following two categories:

- Representation at the lexical level, modelled through physical instances of available virtual
interaction objects (*Higher Level Presentation*).

- Representation at the lexical level, not modelled through instances of virtual interaction objects (*Lower Level Presentation*); lower level presentational constructs are required (i.e. a conventional visual graphics packet, such as GKS, CORE or PHIGS and lower level facilities of a windowing system, such as Xlib for X WINDOW SYSTEM, etc).

This type of conceptual differentiation for data representation methods has emerged as a practical necessity, due to the diversity of the scope of physical facilities supplied by the various lexical technologies. Also, such separation reflects the different effort required in order to specify the handling of representation, between the utilisation of interaction objects and the employment of lower level physical constructs. For higher level presentation, formal specification is naturally included in the description of higher level dialogue; Dialogue Agents control collections of virtual interaction objects (through Representation Agents) which may be associated with particular internal data. Notification
Fig.4.3.1.2.2: Higher Level Representation Management

for updates and management (e.g. refreshing affected items) is transparent to the designer and is handled automatically by the involved Dialogue Agents (see Figure 4.3.1.2.2.).

Lower level presentation is handled in a different and more complicated manner. Obviously, a detailed description of the presentational aspects is required for the target display environment. Also, the various provided interactive facilities, in order to access presented information, must be defined through the lower level dialogue specification method; an explicit notification to the update propagation mechanism (Control), must be provided by the designer, when an interactive modification of information by the user is formally expressed at the lower level dialogue specification. Moreover, the designer needs to specify the translation logic, from external structures to internal data and vice versa, for the Control part of the model, which at run-time is realized via lower level Representation Agents (see Figure 4.3.1.2.3.). For simplicity, the lower level dialogue control is drawn as a separate component in order to emphasize explicit incoming notifications to the global notifier; however, from an implementational perspective, the final (lower level) dialogue management module incorporates representation control functionality.
Various structural and functional similarities are illustrated in Figure 5 and Figure 6 concerning the
general state automata which are dedicated to the higher and lower level representation tasks. Temporal
sequencing of functions is indicated via successive numbering. However, a fundamental operational
difference between these two methods is related to the way externally caused updates are generally handled:

- Higher level representation: upon arrival (at the global notifier) of a message reporting an
  external update, the physical mapping would have already been realized on the display;
  physical interaction objects take care automatically of the various lexical concerns (user
  actions and corresponding effects). This is a type of \textit{a-posteriori} notification. However, in case
  of multiple views of application objects (presentational multimodality), dependent updates
  must be explicitly re-mapped in order to preserve consistency of representation.

- Lower level representation: the designer must describe the various user actions causing an
  external update and subsequently request for appropriate mapping through an explicit
  notification to the global notification agent; side effects of updates are also calculated and
  distributed appropriately to the involved representation agents. The previous separation of
  actions and effects does not imply any fundamental notational distinction between input and
  output expressions at the lower level of dialogue specification method. Figure 4.3.1.2.3.
  outlines a functional property of the underlying control, not of the supplied specification
A formalism is mentioned, which is a type of a-priori notification of updates, requiring mapping of the original physical effects (we call this deferred mapping).

Maximum flexibility is provided to the designer regarding the combination of higher and lower level representation methods; for the same application objects, adoption of different representation levels for each display environment is allowed. This means that the identical species of internal data may be mapped via an interaction object in the non-visual environment (e.g. higher level representation), while a combination of lower level lexical structures can be applied in the visual environment (e.g. lower level representation). For instance, consider a special purpose graphic editing program, where the user is allowed to construct structured figures using higher level drawing primitives (e.g. hypothetical primitives such as connectors, containers, separators, hierarchies, etc). Assume that the visual dialogue is provided with specific interaction objects by the utilised lexical technology, in order to handle user definition for these graphical entities. The designer may specify virtual interaction objects that correspond to these specific lexical instances, which will also manage representation in the visual display environment. Proper association to internal application oriented objects need to be specified by the designer. Consider that there is an appropriate internal engine in order to transform higher level graphic documents to suitable textual descriptions. Such a requirement is reasonable when related to the application functionality; it is assumed that the application developer will be aware of the fact that the resulting interactive system is to be accessible also by blind users, and would therefore need to provide the means of communicating internally information in either shared presentation independent forms or multiple alternative images (e.g. in our example geometrical and textual descriptions). Such textual description may be provided to the blind user through a non-visual physical interaction object; moreover, each time the sighted user modifies the document, the internal engine is activated and consequently provides the most recent description via the associated non-visual interaction object. Notice that such an internal machine should only recompute textual description for the affected context of the document. This example demonstrates the power and appropriateness of the adopted representation control schema, concerning efficient support for Dual User Interfaces. Internally, such desired functionality is achieved through the implementation of a common Update Notification Agent, which communicates with the control parts of all (PAC oriented) representation agents; all externally (from the user) caused updates are reported to this agent, while the Update Notification Agent must distribute updates originated from either internal (application) or external (user) sources, to the appropriate representation agents involved, in order to refresh data mapping, according to their most recent status (see Figure 4.3.1.2.2. and Figure 4.3.1.2.3.).

### 4.3.1.3. Overview - Structure and Relations

In Figure 4.3.1.3.1., an overview of the interrelations among the required formal structures is portrayed. Arrows indicate inheritance of knowledge concerning formally expressed items. Shared objects and direct messages serve interfacing purposes, as will be explained later on. Also, specifications involving lexical constructs must be provided appropriately, once for each environment (e.g. Instantiation Logic, Lower Level Dialogue & Representation and Exported Lexical Concepts / Direct Messages). There are several additional significant properties concerning the proposed specification scheme, some of which are identified below:

- Following the definition (by the designer) of a particular set of virtual interaction objects, further exploitation for the construction of various (Dual or conventional) User Interfaces is allowed (reusability). Multiple collections of virtual interaction objects can be specified, but only one may be utilised during the development cycle of a particular User Interface.

- Specification of instantiation logic needs to be supplied only once. However, multiple instantiation schemes can be associated to a single set of virtual interaction objects. This feature allows the flexible provision of various physical images for the same dialogue structure. Moreover, considering that object instantiation is performed during user-computer
dialogue, it is possible to "switch" between various instantiation schemes at run-time, causing an immediate update of the behavioral and lexical attitudes for the instances of the involved virtual interaction objects. Also the same virtual set may be instantiated appropriately in various platforms / environments, allowing portability of original formal specifications that are based on virtual interaction objects.

At the lower level dialogue specification method, the provision of universal formal names is allowed, concerning the exported (from the utilised lexical technologies) lexical concepts. In this way, direct applicability of original formal specifications along the various lexical environments is accomplished, because the same naming conventions are employed. Evidently, in order to ensure that the resulting dialogue would be meaningful in its various "physical pictures" (i.e. the mapping of the various notational dialogue constructs to the corresponding physical entities for each lexical technology), there should be a consistent and rational correlation of similar physical concepts (of different lexical technologies) to a universal (shared) identifier. For example, an identifier "menu" may be associated with various physical objects among different environments; a considerable homogeneity of its various lexical and behavioral structures is required and, furthermore, these presentational entities should consistently conform with the concept represented by its name.
Fig.4.3.1.3.1. : Overview of the Dialogue Specification Scenario
A Brief Example

Consider that the goal is to accomplish the development of a Dual User Interface for a particular operating system (e.g. a shell). It is assumed that the designer decides to provide WYSIWYG facilities for the visual environment, while a conventional text-based command interpreter is provided in the non-visual form; apparently, both users still share equivalent underlying functionality. Assume that the utilised lexical technology does not provide any built-in methods for interaction objects in order to cope with point-and-dragging specific interaction; consequently, explicit specification of such lower level dialogue is required. Also, external representation of files in the visual environment is accomplished through specific interaction objects, while the application interface is established in terms of shared objects corresponding to internal files. The sighted user is able to perform some functions of the operating system in a direct manipulation fashion (i.e. delete, copy, rename, execute, etc). Assume that deletion of one or more files is carried out in this form; then the following is a possible scenario, which may be specified by the designer, in order to inform the blind user about the removed files. Within the lower level dialogue, it is specified that shared objects corresponding to deleted files (e.g. interaction objects) should be destroyed. Normally, the Application Component will be notified for this event and subsequently the actual deletion of files will take place; however, it is the responsibility of the application developer to interpret such events appropriately and call such related functions accordingly. Also, destruction of shared objects may trigger appropriate Dialogue Agents, that have been specified by the designer, which will initiate the provision of a message (i.e. through a non-visual interaction object) to the blind user informing about the identities of the deleted files. It should be mentioned that Dialogue Agents will be notified prior to destruction; according to the adopted management policy, distribution of object identity to interested destinations (i.e. those that should be informed about destruction) precedes actual destruction. In order to support similar specifications, appropriate notational facilities should be provided for utilising interaction objects and also for lower level lexical constructs.

4.3.2. Integrating and Interfacing Lexical Technologies

An integrated methodology is required for the introduction of various visual and non-visual lexical technologies, which will provide the means of handling physical interaction. Currently, there is a considerable pluralism concerning the domain of visual lexical technologies, the vast majority of which adopts the interaction object paradigm. However, due to the absence of similar User Interface technologies for non-visual interaction, lower level interactive facilities need to be allowed as well (e.g. device oriented input, instead of the logical abstraction offered through interaction objects; however, a non-visual User Interface toolkit based on interaction objects is considered significantly more efficient and appropriate for the lexical level).

Notational integration of a lexical technology is achieved by means of introducing specific lexical constructs to the lower level dialogue specification method. This is achieved through the formal description of such shared exported (from the technology) lexical concepts and also direct messages, which will consequently form the basis of the communication protocol between the rest of the run-time system and the lexical technology. Also, this method enables the programmatic construction of artificial lexical constructs that are not provided directly (e.g built-in) from the lexical technology. Exported lexical constructs may include interaction objects with associated methods and attributes, various lexical services that should be controlled by the designer, graphic primitives, peripheral devices, various event types, etc. For instance, interaction objects and their attributes might be specified as shared entities through the realisation of intercomponent communication, concerning these objects, via transactions. Specific event types and object methods could be described as direct messages in accordance to a formal method allowing appropriate dispatching at the lower level dialogue (e.g. reaction upon arrival of specific messages from the lexical technology). However, it is likely that the lexical technology may not fit directly to the various communication needs previously
mentioned; an adaptation level is required, which will provide, transparently to the higher level components of the run-time system, access to the lexical services supported by the technology.

The last component (*Lexical Technology Server*, see Figure 4.3.4.1.) should follow the predefined protocol, which is implemented by means of direct messages and transactions on shared objects (see Figure 4.3.4.2.). Consequently, implementation of this module is required for each different lexical technology, in order to translate from protocol oriented constructs to appropriate requests for the lexical technology. This approach is based on a similar method followed in SERPENT [BASS90]. It is possible to specify a universal logical interface to the various Lexical Technology Servers in terms of the exported concepts and the adopted communication protocol, and apply this protocol to every lexical technology being utilised. In order to facilitate such a standardisation of the accessible exported lexical concepts, the implementation of the corresponding Lexical Technology Servers should follow such a specific protocol. In this case, the same dialogue specification may be ported and applied without requiring modifications to every graphical environment for which the required Lexical Technology Server is compatible to the universal protocol. It should be mentioned that this approach is in contrast to SERPENT [CMU91] where, upon integration, the application of the naming conventions originally adopted by the lexical technology is encouraged.

### 4.3.3. Enroling the Application Component

Even though the concept of Dual Interfaces is not expected to affect seriously the decision for an appropriate methodology, facilitating the efficient communication with the Application Component, a harmonization with the various specified architectural characteristics is required, concerning the Dual dialogue management. Originally, UIMS technology had adopted a simple protocol based on function calls and parameter passing. Application was usually treated as a semantic server and primitive data exchange was allowed through a restricted synchronous method. Afterwards, blackboard and shared memory techniques were applied by means of shared conceptual objects and asynchronous communication.

In the past, the ultimate goal was to achieve a modular separation between interactive and computational components; on the contrary, today, it is the provision of a highly efficient application interface enabling a broad communication channel, that is considered critical even though reduction of intermediate interconnections to the dialogue and presentation layer may be required. The supported method is based on a bidirectional communication channel, in terms of shared objects. Direct exchange of messages is also allowed. Semantic enhancement is easily achieved by means of exported and appropriately adapted application oriented objects. The principle of providing a type of shared semantic / conceptual knowledge, by means of exported internal entities, is based on original ideas found in [COUT91], and similar ideas found in [DUCE90], where the notion of an Application Conceptual Interface via shared Conceptual Objects is introduced. The underlying transaction management and direct communication package is also applied for the establishment of communication interfaces between the various components of the architecture (i.e. between Lexical Technology Server and the rest of the run-time system). The designer must provide formal descriptions of objects being exchanged between the communicators (for communication established either directly, or via shared space).

### 4.3.4. The Proposed Run-Time Architectural Scenario

In Figure 4.3.4.1., a general functional model is illustrated, regarding the run-time system, which reflects the various architectural properties that have been introduced in the context of Dual User Interfaces. Apparently, a considerable number of intermediate interfacing levels is proposed, by the provision of a common inter-component communication mechanism (e.g. the underlying package which implements the low level aspects of either the Application Interface or the interface with the
Lexical Technology Server). This mechanism, as previously described, is based on the concept of shared typed objects (transactions on shared space) and direct exchange of messages (direct communication).

Following this architectural structure, fine grain control is achieved through distributed multilevel dialogue management, according to the earlier described differentiation of the various interactive phenomena to higher and lower level classes. This direct functional mapping of the adopted conceptual model has been considered the most efficient approach in order to meet the needs of modular run-time organization for Dual User Interfaces.
In Figure 4.3.4.1., it is indicated that the nature of dialogue management, for each level of interaction, is decided by the formal specifications provided by the designer. However, the notation does not imply that at run-time, the original specifications are internally being consulted for carrying out dialogue control tasks; actually, dialogue meta-descriptions, appropriately derived from provided specifications, are utilised at run-time for similar purposes. The various components that communicate via the supported interfacing mechanism with other modules, must be generated as independent processes of the operating system. The only exception is the integration of Technology Servers with corresponding Lexical Technologies (that can be faced as conventional programming libraries) where communication is achieved through function calls (procedural interface); these modules will eventually constitute a single process.

![Diagram of Intercomponent Communication Method]

**Fig.4.3.4.2. : Realisation of Intercomponent Communication Method**

In accordance with the transaction processing mechanism, messaging techniques have been incorporated as a complementary method considering that in various communicating tasks direct exchange of messages is more easily and efficiently conceived (see Figure 4.3.4.2.). Hence, when dealing with the design of appropriate tools aiming to allow specification of intercomponent conversation protocol by the designer, it is very important to support a natural mapping between the particular communication needs and the corresponding supplied formal description technique. For instance, in case that exchange of messages must be modelled, similar message-oriented specifications must be allowed, even though description by means of transactions could be achieved, but not in such a straightforward manner. Consequently, the adopted method employs both message-based and transaction-based (e.g. shared memory) communication methods. Also, necessary high level specification facilities are required in order to enable the designer to efficiently cope with the various communication requirements through flexible utilisation of techniques based either on shared entities or on direct messages without involving implementation oriented concepts.

From lower level dialogue, communication with application is allowed, via the common for all components interface. This feature has been incorporated, in order to achieve efficient handling of semantic oriented actions that are required from the lower level dialogue control and also direct access.
on the internal structures mapped via lower level presentation; the higher level module could be utilised as a mediator, but slower performance would result due to the additional levels for communication and interpretation of messages. Considering that by introducing an alternative display environment, a natural bandwidth increase is expected at this level, the adopted approach is more appropriate because it does not deteriorate the "communication bottleneck problem", related to the provision of fast semantic feedback. However, this does not eliminate the problem.

Instantiation logic (from virtual objects to the corresponding physical entities) has been incorporated to the lower level dialogue modules. The communication with the higher level module is via presentation independent messages and shared concepts (i.e. request for object instantiation / destruction through transactions on shared objects, request the refreshing for mapping of semantic information for internally caused updates, request the execution of methods associated to objects, notify for externally caused updates, etc). Concerning representation management of application oriented information, translation logic is either included in the higher dialogue control (for higher level presentation) or in the lower level dialogue control (for lower level presentation). For simplicity, such issues are not reflected in the overview of the run-time architecture (see Figure 4.3.4.1.).

4.3.5. An Example of a Dual Interface

A simple hypothetical example will be described, which demonstrates the main principles of a Dual interaction paradigm. Consider that the goal is to achieve Dual reproduction of an interaction object which consists of a list of textual editable field entries (e.g. a form object). A possible reproduction scenario is outlined in Figure 4.3.5.1. The physical level of interaction, for each environment, is managed by the utilised lexical level technology. However, propagation of updates is the responsibility of the UIMS. A brief description of such a Dual dialogue scenario is given below.

Non-visual environment

An auditory symbolic representation of the editable fields is provided through distinct sound effects which are separated both spatially (in 3D space) and temporally (the sound utterances are provided sequentially, between specific time intervals). However, at this level, there is no physical reproduction of the text enclosed in each field. Input, in the non-visual environment, is possible via speech, 3D-gestures, typing and 3D-pointing, while speech and tactile output are also supported. The user may point via a data glove (in 3D space) to a particular field, while in the mean time the word "this" is to be uttered with a head-mounted microphone; speech recognition is required for a set of keywords. The lexical technology automatically provides the interpretation of the textual field in Braille on the tactile module, while providing the following acoustic feedback through the headphones: "field ... has been selected, please review text in the tactile module". The blind user is able to update the contents of the field through textual input from either the keyboard, or the microphone; in this case (content modification) the visual representation is automatically refreshed according to the new field value.

Visual environment

Physical interaction with a sighted user is easily managed by the toolkit. However, in case that the sighted user causes any update to the field values, propagation to the non-visual representation is required. This could be possible in the following manner: the volume of the sound effect which corresponds to the recently modified field is increased, and returns to its normal (i.e. predefined by the dialogue designer) state only after the blind user has reviewed the field in question with the new value; while the blind user is reviewing the specific update field, perception of the update is straightforward, and there is no need for any other non-visual physical stimulus.
Fig. 4.3.5.1. : Example of a Dual Interface
4.4. Analytic System Design

The proposed design formalism which addresses the specification requirements of the higher-level dialogue is analytically described below. The syntactic aspects of the associated constructs will be presented in a grammar-oriented form (italicised words correspond to non-terminal grammar symbols while bold-faced words represent terminal symbols - tokens), in accordance to parse methods (e.g. translation schemes) and code-generation techniques (with target language being C and C++). Also, the various run-time management algorithms will be described. According to the proposed approach, the eventual module for the higher-level is mostly code-generated from the compilation of the corresponding dialogue description. The proposed method is based on the construction of an efficient and reusable skeleton of the higher-level control process that is glued appropriately together with the various code segments generated upon compilation of the formal dialogue structures in order to form the higher-level run-time package. Also, various meta-structures produced at the translation phase are also inquired during run-time. Additionally, intuitive proofs of completeness are also outlined for some language properties, and theoretical paradigms (scenarios) are investigated as a test-bed for expressive power issues.

4.4.1. Conventional Programming Constructs

Common programming constructs concern the subset of the provided expressive facilities which is similar to the concept of particular building blocks found in various procedural programming languages (e.g. Pascal, C, C++, etc). Even though they are distinguished from the notational techniques that are specific to dialogue description, they have been appropriately formalised so that a flexible and uniform integration with the various interface specification primitives can be achieved in a common formal framework. The major supported programming constructs are data-types, variables, statements, and functions (following the C-philosophy, but may also be called procedures or methods in other languages). Some syntactic aspects are similar to Pascal while the semantics are mostly C-oriented. It should be mentioned that some constructs require an understanding of some language properties that are subsequently discussed. This is due to the fact that many inter-relationships and inter-dependencies exist between the constructs of the language and it is difficult to establish an optimal way of presenting the design results. Actually, the whole process was a highly iterative procedure with many links back-and-forth between the various design tasks. Nevertheless, in case that references are required to some particular constructs which are studied later, a rough description will be initially provided, while the relative issues will be revisited in more detail in the corresponding dedicated chapter.

Definition of data-types & declaration of variables

In the provided formal scenario, the declaration of variables is separated from the definition of the corresponding types (for user defined data-types) for clarity. For example, in C it is possible to declare directly variables within the definition of structures; even though the later is not supported in the proposed language, it does not affect its expressive power. However, in a particular implementation of the proposed framework the convention that is adopted in most programming languages may be supported. Also, even though data-initialization rules are not included in the presented formalism (for expressive economy), they are considered part of our formal model (formal rules for describing data-initialization may be found to the syntax description of the standard forms of most programming languages).

```
DataTypeDefinition    : LocalTypeSynonyms
                      | Enumerated
                      | Structure
                      | ForwardStructureDecl
                      | VirtualObjGenesis
```
LocalTypeSynonyms : TYPE TypeSpecifier Variable ;

The non-terminal Variable will be finally reduced to an IDENTIFIER which will consequently be considered as the synonym of the type specified through the combination of the TypeSpecifier and Variable grammar rules. This construct is similar to the typedef facility of C, and it is a trivial task to define a translation scheme to emit C-code for the corresponding typedef statement. For instance, TYPE integer Array10[10] defines a new type recognised through the identifier Array10 and is equivalent to the C-code typedef int Array10[10];.

Enumerated : ENUM IDENTIFIER = { IdentList }

IdentList : IDENTIFIER
  | IdentList , IDENTIFIER

Structure : STRUCT IDENTIFIER
  {
    VariableDeclarations
  }

ForwardStructureDecl : FORWARD STRUCT IDENTIFIER

VariableDeclarations : VariableDeclarations DeclStmt
  | DeclStmt

DeclStmt : Qualifier TypeSpecifier VariableList ;

Qualifier
  : PRIVATE
  | SHARED
  | EXTERN_SHARED

Forward structure declarations have been introduced to cope with mutually recursive data structures. PRIVATE qualified variables are only visible within the dialogue specification modules. Also, PRIVATE is the default specifier for variables. SHARED qualified variables are made accessible (automatically) also by the application component and the lower-level dialogue (actually, there are two different lower level control modules, visual & non-visual respectively, however, for expressive convenience we may also use the term lower-level dialogue to refer to both), in terms of a shared typed object. Only global variables may be shared (i.e. static variables). It should be noted that this type of shared variables are static shared objects and they cannot be destroyed. They have been introduced in order to provide a means of transparent sharing through built-in programming constructs, rather than fulfilling the basic communication requirements (as it will be explained later-on, there are specific dedicated constructs to cope with shared-memory and object sharing issues). With this respect, shared variables may participate in various formal schemes (such as dependencies). A technique demonstrating how SHARED variables may be handled is outlined:

0 The variable is declared as shared only in one module, for instance, at the higher-level dialogue. In this case, automatically three (3) instances of a shared object are created, both of type identical to the base type of the shared variable and with the identifier that is used within the original declaration (variable implicitly exported). These instances are exposed on the various implementations of shared-space used to provide the intercomponent communication with the two lower-level modules and with the application component. In order to enable
notational access, such shared variables must be declared as `extern_shared <variable>` (EXTERN_SHARED terminal symbol) which would inform the compiler that such objects already exist in the shared-space (because the owner module takes care of initial instantiation), and that all references (read / write) should be translated appropriately to communication requests of the underlying package upon code generation.

```
TypeSpecifier : BuiltIn
| UserDefined
;

BuiltIn : INTEGER
| CHAR
| BOOLEAN
| REAL
| STRING
| VO_ID  /* virtual object id */
| DA_ID  /* Dialogue Agent id */
| SHARED_ID /* shared object id */
;

UserDefined : IDENTIFIER
;

VariableList : VariableList , Variable
| Variable
;

Variable : IDENTIFIER
| Array
| Pointer
| ArrayOfPointers
;

Array : IDENTIFIER BracketCollection
;

BracketCollection : [ CONST_EXPRESSION ]
| BracketCollection [ Expressionconst ]
;

Pointer : IDENTIFIER StarCollection
;

StarCollection : *
| StarCollection *
;

ArrayOfPointers : StartCollection Array
;
```

Notice that the supported types are scalar, arrays of scalar types or structures, and arrays of pointers with an arbitrary depth of indirection. This is only a representative set of supported types which is provided here for completeness; proper extensions are allowed in order to enrich the programming-oriented expressive capabilities and constructs. The VO_ID (virtual object identifier) and DA_ID (Dialogue Agent identifier) terminal symbols, which represent two different built-in data-types, have been introduced to cope with the problem of handling notationally various dynamic instantiations of the same construct, which cannot be distinguished at run-time only by name (this problem will be studied in detail in the context of virtual interaction objects).

**Statements & Functions**

The range of statements that are supported is similar to those found in many programming languages.
Such constructs are considered very flexible and powerful for a dialogue specification language. Some of the supported semantics are quite low-level and require an understanding of the basic programming issues. However, our methodology aims to support a kernel of higher-level formal items in accordance to some additional appropriately selected lower-level programming facilities to enable more difficult and complex problems to be adequately addressed. For instance, functions are not part of the fundamental constructs which were especially introduced in order to fulfil the requirements of supporting the development of Dual Interfaces, however, they contribute seriously to the power and flexibility of the original method in certain situations that reusable functional components are required at the dialogue level (sometimes it is necessary to build short procedures within the dialogue-level due to performance problems if similar trivial tasks are always delegated to the application component).

```
Stmts : Stmts Stmt |
       Stmt ;

Stmt : Assignment |
      FunctionCall |
      Conditional |
      Loop |
      Case |
      DeclStmt |
      MsgSend |
      CreateSharedObject |
      UpdateSharedObject |
      DestroySharedObject |
      InstantiateDialogueAgent |
      DisableDialogueAgent |
      InstantiateObject |
      SetMethod |
      BREAK |
      CONTINUE |
      DONE |
      TERMINATE |
      Compound ;

Assignment : Lvalue = Expression ;

FunctionCall : IDENTIFIER function name ( ExpressionList ) ;

Conditional : IF (BoolExpression) THEN Stmt |
              IF (BoolExpression) THEN Stmt ELSE Stmt ;

Loop : WHILE (BoolExpression) DO Stmt ;

Case : CASE (Expression) OF |
       [ CasePairs ] |
       ELSE Stmt ;
```
CasePairs : CasePairs CasePair
   | CasePair
   |
CasePair   : Expression_const: Stmt
   |
ExpressionList : ExpressionList , Expression
   | Expression
   |
LvalueList   : LvalueList Lvalue
   | Lvalue
   |
FunctionDef : TypeSpecifier IDENTIFIER ( VariableDeclarations )
   Compound
   |
Lvalue      : VirtualObjectContent
   | ProgLvalue
   |

An Lvalue is either a VirtualObjectContent (accessing a local variable of a virtual interaction object) or ProgLvalue (conventional lvalues as in natural programming languages). The way the local structures of virtual objects are notationally referenced will be presented under chapter-4.4.5., while ProgLvalue is not further analyzed, considering that it is a common programming concept independent of our specific methodology.
4.4.2. Dependency Schemes and Constraint Satisfaction

Dependencies concern the declaration of constraints among objects (such as virtual interaction objects or variables) which must be satisfied at run-time. The designer may only draw a formal picture of the relations that must hold among objects during user computer-dialogue, without any concern on run-time management and constraint resolution issues. Constraints are remarkably efficient syntactic constructs, which appropriately address various practical expressive problems related to the specification of direct manipulation interfaces. While the number of dependency relations grows linearly, the complexity of resolution algorithm is non-linear (this is due to the fact that a single dependency equation usually produces multiple dependency links - proportional to the number of the engaged objects). In the proposed scheme, only unidirectional constraints are supported (simple forms of bidirectional constraints can be specified by applying pairs of unidirectional constraints). Constraints are enabled (i.e. they must be satisfied by the run-time module) for the period of time that an associated enabling condition is true. An efficient constraint satisfaction algorithm has been designed, both to resolve dependencies and to recompute affected enabling conditions, which must be called each time particular objects which are engaged in dependencies are modified. Constraint satisfaction is applied upon notification of updates and generates the list of totally affected objects, after re-computing their new values. Next follows a formal scheme for defining dependencies.

```
ConstraintSpecs : ConstraintSpecs ConstraintSpec |
                 ConstraintSpec;

ConstraintSpec : Lvalue :
                   [ DependencyPairs DEFAULT : Expression ];

DependencyPairs : DependencyPairs , DependencyPair |
                   DependencyPair;

DependencyPair : EMPTY |
                BoolExpression ? Expression;
```

**Example-1**

```
1: s : [ 
2:  object moving ? u * t 
3:  default : 0 
4: ] 
5: 
6:  [VideoVolumeLabel].label : [ 
7:  video on : [VideoVolumeValuator].value + 10 
8:  not video on & video incoming : appl_volume 
9:  default : CLOSED 
10: ]
```

In this scheme, consistent type conversion is required between the various types of expressions which are engaged in dependency pairs (source type) and the type of the dependent object (target type). Considering that all these types are completely known at compile time, the conversion method is simplified and proven existing techniques from compiler construction theory may be applied. In Example-1, a simple scenario of two dependency equations is presented. In the first dependency, the s (space) object depends on u (velocity) and t (time) through the equation u * t. Roughly speaking, whenever one of these objects is updated, the value of the condition video_on is re-computed. If the condition is found to be enabled, then the s object is assigned the value of the product, otherwise the default value 0 is assigned. Also, the same re-evaluation procedure must be triggered each time...
video_on is modified. Consequently, not only the expression portions of the equation are tested for updates, but the conditions are appropriately re-evaluated as well. In Figure 4.4.2.1., the structure of the formal scheme for dependencies is outlined.

Fig.4.4.2.1. : Structure of a Dependency Equation

The algorithm for constraint satisfaction inquires the object dependency graph that is generated during compilation. This dependency graph is constructed based on the various dependency equations that have been specified. A formal definition of dependency relations follows:

- Let X be an lvalue type object (it has a permanent storage in our program) that depends upon expression E. Then if Y is another lvalue object where Y is in E, then X is said to depend on Y, X <- Y. Also, if Y is part of the boolean expression, then X <- Y.

- Relation <- is reflexive and transitive, and taking the reflexive transitive closure of relations, an object X is said to depend on another object Y, X <-* Y, if it is (X <- W and W <-* Y) or (X <- Y).
**Algorithm to Compute Order of Computation for Updates**

- Let $U_1, \ldots, U_n$ be the updated objects, for which the constraint satisfaction algorithm has to be applied, in order to compute additional updates.

- Let $S_1, \ldots, S_m$ be sets of objects. At the end of the algorithm, the computation of updates will be $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m$, in order to produce a consistent computation. Initially it is $S_1 = \{U_1, \ldots, U_n\}$, $S_2 = S_3 = \ldots = S_m = \{}$.

- Let $P$ and $Q$ be sets of objects.

  For each set $S_i : i=1, \ldots, m-1$ Do
  
  - $P=S_i$, $Q=\{}$.
  
  For each object $X$ in $P$ Do
  
  - Let $O_1, \ldots, O_k$ be objects that have dependencies directly (of $\leftarrow$ type) with the object $X$ in $P$. Set $Q = Q \cup \{O_1, \ldots, O_k\}$.
  
  If ($Q = \{}$) Then
  
  - Set $Depth = i$.
  
  - Break the loop.

  $S_{i+1} = S_i \cup Q$.

  
  For each $S_i : i=Depth, \ldots, 2$ Do
  
  - $S_{i-1} = S_{i-1} - (S_i \cup S_{i+1} \cup \ldots \cup S_{Depth})$.

  
  Now $S_1, \ldots, S_{Depth}$ are disjoint, and this ordering defines a consistent computation of updates between objects. Updates within the same set may be computed in any order. Each object will only be computed once.

- From left to right recompute for each object only these expressions that depend directly on updated variables. The algorithm ensures that with this computation, if an object $o_1$ depends directly on another $o_2$, then object $o_2$ will be computed before $o_1$. 
Algorithm to Resolve Constraints

Let \( S_1, \ldots, S_d \) be the sets defining the computation order for updates (the output of the previous algorithm). Let \( Q = <S_1, \ldots, S_d> \) be an ordered list composed of these sets, where elements of the same set may be placed in any relative order, and if for each \( X_i \) and \( X_j \) it is \( i < j \) if \( X_i \) in \( S_q \) and \( X_j \) in \( S_l \), \( j < l \), and \( i = j \) if \( q = l \).

Let \( (C, E) \) represent condition-expression pairs which compose dependencies. Let \( \text{NoDependency} \) be a boolean variable, initially True, that defines whether for an object a dependency has been found at the end.

For each object \( X \) in \( Q \) Do

\[
\begin{align*}
    \text{If} & \quad (X \text{ was assigned by a } (C_i, E_i)) \quad \text{Then} \\
    \text{If} & \quad (C_i \text{ does not depend on any } Y \text{ before } X \text{ in } Q) \quad \text{Then} \\
    \{ & \quad (E_i \text{ depends on any } Z \text{ before } X \text{ in } Q) \quad \text{Then} \\
    \quad & \quad X = \text{re-evaluate}(E_i). \\
    \quad & \quad \text{NoDependency} = \text{False}. \\
    \} & \quad \text{Else} \\
    \{ & \quad C_i = \text{re-evaluate}(C_i). \\
    \quad & \quad (C_i = \text{True}) \quad \text{Then} \\
    \quad & \quad (E_i \text{ depends on any } Y \text{ before } X \text{ in } Q) \quad \text{Then} \\
    \quad & \quad X = \text{re-evaluate}(E_i). \\
    \quad & \quad \text{NoDependency} = \text{False}. \\
    \} & \quad \text{Else} \\
    \{ & \quad \text{Let } \{(C_1, E_1), \ldots, (C_n, E_n)\} \text{ be the dependency pairs of } X \text{ having a condition part dependent on an object } Y \text{ before } X \text{ in } Q. \\
    \text{For} & \quad \text{each } (C_i, E_i) \quad \text{Do} \\
    \{ & \quad C_i = \text{re-evaluate}(C_i). \\
    \quad & \quad (C_i = \text{True}) \quad \text{Then} \\
    \quad & \quad (E_i \text{ depends on any } Y \text{ before } X \text{ in } Q) \quad \text{Then} \\
    \quad & \quad X = \text{re-evaluate}(E_i). \\
    \quad & \quad \text{NoDependency} = \text{False}. \\
    \quad & \quad \text{Break}. \\
    \} & \quad \text{Else} \\
    \} & \quad \text{Else} \\
\end{align*}
\]

If \( \text{NoDependency} \) Then

\[
X = \text{re-evaluate}((\text{Default}(X))).
\]
Example-2

For1:

<table>
<thead>
<tr>
<th>Set</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>{w}</td>
<td>P = S1,</td>
<td>Q = {y, z, q}</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>{y, z, q}</td>
<td>P = S2,</td>
<td>Q = {x, y, z}</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>{x, y, z}</td>
<td>P = S3,</td>
<td>Q = {x, y}</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>{x, y}</td>
<td>P = S4,</td>
<td>Q = {y, z, q}</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>{x}</td>
<td>P = S5,</td>
<td>Q = {}</td>
<td></td>
</tr>
</tbody>
</table>

Depth = 5

For2:

<table>
<thead>
<tr>
<th>Set</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>{w}</td>
<td>P = S1,</td>
<td>Q = {y, z, q}</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>{y, z, q}</td>
<td>P = S2,</td>
<td>Q = {x, y, z}</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>{x, y, z}</td>
<td>P = S3,</td>
<td>Q = {x, y}</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>{x, y}</td>
<td>P = S4,</td>
<td>Q = {y, z, q}</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>{x}</td>
<td>P = S5,</td>
<td>Q = {}</td>
<td></td>
</tr>
</tbody>
</table>

Consistent Order for Computation of updates is:

w -> qw -> zw, q -> yz, w,q -> xy, z

Fig.4.4.2.2. : An Example Dependency Graph

In Example-2, a simple dependency graph is presented, and the various passes of the algorithm for the generation of the computation order for updates is presented, considering that object \( w \) is modified. An interesting subject is related to the way the dependency graph is incrementally build (structured) at the compilation phase. Another problem is how individual nodes of the graph correspond to
particular objects of the original dialogue specification (for instance, programming variables or virtual
interaction objects). Also, a code generation schema and integrated management scenario is required
to completely cover the various aspects of constraint satisfaction. The following are the methods and
rules that must be followed in order to handle constraint satisfaction. In Figure 4.4.2.3. the component
of the dependency graph incorporated due to the equation is presented.

I. Let \((C_i, E_i)\) be a condition-expression pair contributing in a particular dependency scheme. Let
also \(T = \{X_1, \ldots, X_n\}\) the set of objects involved in either the condition or the expression. Then
for each such \(X_i\), if there is no corresponding node in the dependency graph, establish a stand-
alone node. Let \(Y\) be the object at the left part of this dependency, then for each node of the
dependency graph corresponding to a \(X_i\) for each condition-expression pair, establish an edge
from \(X_i\) to \(Y\).

\[
\begin{align*}
X_1 & \rightarrow Y \\
X_n & \rightarrow Y \\
C_1 : E_1 & \\
C_i : E_i & \\
C_m : E_m & \\
E_{\text{default}} & \\
\end{align*}
\]

Fig.4.4.2.3. : Subgraph Corresponding to a Dependency Equation

II. Let \(<\text{object}\>\) denote the object being involved in the left part of the \(i\) dependency (order
dependencies as they met upon parsing). Then consider as \(Y\) object of the \(I\) rule the object
named \(i_<\text{object}>\). Generate the function \(\text{assign}_{i_<\text{object}>}(\text{<value parms>})\) if called at
run-time will assign at the internal representation of the lvalue for \(<\text{object}>\), as involved in
the \(i\) dependency, the value(-s) passed as actual arguments. For instance, if \(\text{myvaluator.val}\)
is involved, the \(<\text{object}>\) is \(\text{myvaluator}\), while \(\text{assign}_{i_<\text{myvaluator}}(\text{val})\) will assign \(\text{val}\) to
the internal representation of \(\text{myvaluator.val}\) (notice that when objects are involved on the
right-side of a dependency - they affect other objects - then they are used with the identifier
of the variable, that is the actual \(<\text{object}>\)).

III. Let \(D = \{ X | X \text{ is involved in a dependency } \}\). Then the dependency graph \(G\) consists of all
objects in $D$ and exactly of these objects. Each object in $D$ is a node in $G$. Consequently, if at run-time an object changes, then the constraint propagation algorithm is applied only if $Y$ is in $D$.

IV. For each dependency $i$, after performing the rules I and II, if $<\text{object}>$ is node a node in $G$, then make a new node $<\text{object}>$ and put an edge from $i_<\text{object}>$ to $<\text{object}>$. According to this rule, if the total number of dependencies is $N$ then $N$ additional edges are added to the dependency graph $G$. These edges are necessary in order to connect updates of the lvalue expressions of an object to the actual object itself (formally $<\text{object}>$ is never modified as a separate object - it is met as $i_<\text{object}>$).

V. For each dependency $i$, consider the following sets, that specify the involvement of objects in dependencies:
- $C_{ij} = \{ X | \text{condition } C_{ij} \text{ depends on } X \}$.
- $E_{ij} = \{ X | \text{expression } E_{ij} \text{ depends on } X \}$.
- $E_{id} = \{ X | \text{default expression for } i \text{ dependency depends on } X \}$.

VI. For each condition and expression of a dependency, consider the generated corresponding function that when called at run-time evaluate that condition and expression. The code generated function contains appropriate references to the internal representations (at run-time) of the involved objects.

VII. By the application of rules I,..,VI upon translation and by the application of the various run-time algorithms previously described the constraint satisfaction component is completed.

In Figure 4.4.2.4. an appropriate data-structure skeleton is presented conforming with the rules I through VI. This data structure is accessed by the constraint satisfaction algorithm, for instance, to check whether an object depends on another either for re-computation of a particular condition / expression or in order to set at run-time the value of an object. The various fields of this data-structure are constructed during compilation according to the various dependency equations being parsed. The rules previously listed, can be translated to operations on this structure; for example, adding an arch from $X_{ij}$ to $Y$ for $(C,E_i)$ is equivalent to inserting the object $X$ in to the set $E_{ij}\_obj\_set$ of the data structure for object $Y$. Clearly, objects that do not participate as dependents on a dependency equation do not have such a data-structure.
Fig.4.4.2.4: Data Structure Skeleton for i-th Dependency Equation for <object>
4.4.3. Methods for Objects & Handlers for Communication Events

Methods

A *method* is related to virtual interaction objects (the basic interactive entities at the higher-level) and can be faced as the code segment to be executed upon the detection of specific logical events originated from the physical layer. With each class of virtual interaction objects, multiple classes of methods may be associated; the architecture is open concerning this feature. A method is visible within the higher-level only if it has been exported (different naming conventions that the original may be adopted) by the lexical technology during the platform integration phase. Following this approach, maximum flexibility is provided in terms of the facilities enabling the introduction of various method types corresponding to notification for arbitrary, possibly artificial, object-specific logical events. Normally, only virtual objects corresponding to physical objects which allow user-input may have methods (however, it is not considered of primary importance to support semantic checking for such inconsistencies). An instance of a method class implements a specific behavior (in terms of a list of statements) for this specific class.

A method may be associated with a virtual interaction object either statically (e.g. during method specification; method instance is tight to a fixed object - *bounded methods*) or dynamically (e.g. via a statement; a method may be related with various objects - *free methods*). In the later case, the method that is associated with a particular virtual object may be determined at run-time, while in the former the associated method is always known at compile-time. However, it is possible that the designer may override through a statement a method dynamically. Consequently, the run-time control must implement execution of the appropriate instance. Also, the second category of methods may be reused among various interaction objects. The syntax of the provided formal language for describing methods and an relevant code generation issues will follow.

**BoundedMethod**

: METHOD IDENTIFIER method class IDENTIFIER object instance

;    

**FreeMethod**

: METHOD IDENTIFIER method class IDENTIFIER object class

;  

There is a different syntactic context in which these two types of methods can be legally defined. Bounded methods may be specified only within the body of Dialogue Agents, while free methods may be specified either within Dialogue Agents or globally. It should be noted that syntactically these two different categories can be distinguished; any parser generator would have a *reduce-reduce conflict* problem with this set of rules. An appropriate conflict-resolution criterion is to determine the second IDENTIFIER represents a class or an instance. For example, in a YACC implementation of such a parser, these two rules would be unified to a single rule, in which testing of the resolution criterion would be incorporated in the translation schemes.

**Method**

: METHOD IDENTIFIER method class IDENTIFIER class / instance

{ if (IsClass($3))
    CurrentMethod=BOUNDED;
  else
    CurrentMethod=FREE;
}

Compound

;
One problem arising from the introduction of free methods is how the caller virtual object can be referenced. Clearly, it is not possible to reference any specific virtual object because any free method may be simultaneously shared among various virtual objects (of the same class). For this purpose, a special built-in construct Me (keyword) has been introduced, which can be notationally used exactly like the particular caller object (if the syntactic use of Me is not compatible with the formalism of the represented class, a compile-error may be generated). Also, its applicability is limited only within methods and constructors of virtual interaction objects (see chapter-4.4.4.); if used elsewhere, a compile-error may be generated. An appropriate methodology for code generation for application of Me within methods follows.

I. For each distinct free method, reserve upon code generation a pointer, say M, to the internal representation of the referent virtual class (e.g. produce code for the declaration of a pointer to the code generated data-structure pattern that corresponds to the referent virtual class). If necessary, name this object uniquely. At run-time, before a method is called, this pointer is always assigned the address of the caller object.

II. Assume that the particular virtual class C is represented upon code generation with the data-structure _C; let _M_C be of type (*_C) code generated variable for the specific class (pointer to _C structure; the M of rule I). Then, in any method with referent class C, Me is translated to (*_M_C).

Example-1

```
1: Method
2: EnterPressed_method class TextualFieldObject_referent class
3: [Me].Text = "hello, world";
4: ]
```

In Example-1, Me key-word is used to reference the caller object in a free method. However, this notation is also applicable in bounded methods. An appropriate code generation pattern will also follow. From this point, it will be adopted the convention of representing identifiers of functions which are built-in the architecture with the first letter of each different word capitalised (like BuiltInFunction) while identifiers of code-generated functions will be written with a leading underscore, and will be named according to specific naming algorithms. In Example-2, a code generation pattern in C for Example-1 is presented.

Example-2

```
/*
 * enum type to distinguish methods - could be void
 */
1: MethodType
2: _method_EnterPressed_TextualFieldObject_free (void)
3: {
4: _free_EnterPressed_TextualFieldObject_PTR->Text= CopyString("hello,world");
5: NotifyUpdate(_free_EnterPressed_TextualFieldObject_PTRName, "Text");
6: }
```

The previous example demonstrates how usage of Me operator is translated to code. Notice that after the modification of the local variable Text, a notification to the dialogue control is made so that possible dependencies will be satisfied. This may be adopted as the general strategy when objects involved in dependencies are modified within statements:

III. Let L={O1,...,On} be the set of objects that are involved at the right-side of a dependency
equation. If a statement of the form $O_i = \langle \text{Expression} \rangle$ is parsed, then after exactly after the code pattern for this statement, an explicit notification for the update of the internal representation of $O_i$ must be emitted.

The update-notification routines may only append to a dedicated list a structure containing enough information about updates and immediately return control to the caller; after control is returned to the dialogue-control module, at the end of the main-control loop the list of pending updates is checked, and all updates are served by a single application of the constraint satisfaction algorithm. The principle behind this technique is to enhance performance by reducing the total amount of the computational intensive applications of constraint resolution. However, this is not a correct approach because inconsistent results may be obtained. The last argument is proved by the following example:

```
Example-3
1:   z : [                 
2:   x > 15 or [Tremble].value <= x/0.15 ? x*10  
3:   default : 0.1        
4: ]                        
5:                           
6: Method UpdateValue Volume 
7: [                           
8:   x=[Volume].Value;       
9:   y=z*15;                
10: ]                     
```

In the previous example it is clear that due to the specified dependency, after the execution of the first statement the value of $z$ may change, even though there is no statement directly modifying $z$. Consequently, if the constraint solver is not called till the control returns to dialogue control module, the value of $z$ (and in effect of $y$) will be incorrect (in this particular example, $z$ will gain its correct value after the application of the constraint solver). Detailed study and discussion concerning the functional structure of the main-control loop and the overall organisation of the higher-level dialogue control will be provided in chapter-4.4.8. For this specific problem the following rule may be followed during compilation:

IV. When an assignment is parsed for which the modified lvalue participates on the right-side of a dependency equation, append a structure carrying information about this object and its new value to a compile-time list of pending updates, and generate code after the assignment for an explicit notification of a pending-update at run-time. Whenever an expression is parsed which may include objects found on the left-side of a dependency equation, for each such object if there is a dependency <- relation with one or more objects in the update-pending list, then generate code for an explicit call `CallConstraintSolver();` which will cause immediate execution of the constraint satisfaction algorithm. Also, the constraint satisfaction will be always executed (if necessary) after control is returned to the main-loop. The proposed scheme is based on the general principle that decisions which could be taken at compile-time should not be left for run-time. Intuitively, more sophisticated optimisations can be applied on the second technique.

**Event handlers for Communicating Events**

Communicating events are generated due to either exchange / sharing of information or posting of requests between the components. It has been considered of primary importance to provide an efficient and flexible communication technique which would enable complex communicating tasks to be conveniently implemented. However, an appropriate higher-level notational facility is required, in order to enable the designer to express easily the various captured communicating requirements to a compact and understandable description without necessary involvement of particular implementation details.
Such formal description may only visualise the adopted conceptual model so that different implementations of the underlying support mechanisms can be realised.

Communicating events may be originated either from shared-space or from message channels. These two categories are treated slightly different in the proposed formalism. The necessary formal constructs of the relevant formal language are provided in terms of a basic scenario, by demonstrating the fundamental principles that must be supported; undoubtedly, this scheme may be further enhanced either syntactically or semantically. There are the following specific classes of communicating events:

- **Messages**
  A message is an information entity sent by one communicator to others, travelling through the message channel, waiting at the receiving-end until being finally received. Multiple messages may be waiting simultaneously into the channel, in which case the order of receipt is totally random. Messages are appropriate for sending particular requests to other components, to acknowledge actions or to exchange data-structures that are rarely communicated between the partners.

- **Instantiation of a Shared Object**
  A shared object may be an arbitrary discrete typed data-structure that is exposed by one communicating partner in order to be made accessible by others. Shared-space may be viewed as a shared-bag where multiple occurrences of the same object are allowed.

- **Destruction of a Shared Object**
  A partner may request the destruction of a shared item. In this case, a notification event for disposition of this object is generated and sent to the interested partners. The original model can be extended by incorporating access-restriction (disallow particular partners to have a destruction privilege over a shared object), however, in the proposed approach such additional features are not particularly necessary.

- **Modification of a Shared Object**
  An object may be updated many times during its lifetime in the shared-space. Such updates will be automatically posted to the interested components for further processing. Consider that particular application structures are exported to the higher-level dialogue via shared facilities. Also, assume that the designer describes a presentation scenario via virtual interaction objects. Clearly, notification for updates of application data would allow management of a consistent mapping between the represented information and the internal structures.

```
CommEventhandler : HANDLER_ON CommEvent
    Compound
;
CommEvent : MsgEvent
    ShdestrEvent
    ShcreateEvent
    ShupdateEvent
;
MsgEvent : MESSAGE
    ( IDENTIFIERmsg class, TypeSpecifierdata class, IDENTIFIERbuffer )
;
ShdestrEvent : SHDELETE
    ( TypeSpecifierdata class, IDENTIFIERbuffer )
;
ShupdateEvent : SHUPDATE
    ( TypeSpecifierdata class, IDENTIFIERpassed id, IDENTIFIERbuffer )
;
```
A possible syntax for these types of communication event-handlers has been given. Apparently, there are various syntactic similarities. A message consists of a single typed information packet. During compilation, the TypeSpecifier non-terminal defines the type of IDENTIFIER_buffer that, upon call, will hold the received message. For shared objects, the technique is slightly different. In case of instantiation, an internal id is returned, and is placed on IDENTIFIER_returned_id which is of type SHARED_ID (it is implicitly declared as variable of this type). Also, such an internal identifier is returned each time an explicit instantiation is requested through the CreateSharedObject (non-terminal) statement (syntactic aspects of this non-terminal will also follow). Similarly, explicit modification and destruction (DestroySharedObject & UpdateSharedObject) require such internal identifiers. For notification about destruction or modification via event-handlers, the internal identifier used in shared-space must be passed. What returned, is a copy of this object in the IDENTIFIER_buffer. An example follows which demonstrates the expressive power of message-based constructs.

Example-1

```
1: HandlerOn Message (LIST_USERS StringArrayType thelist)
2: |
3: if (theList[0] = "nouser") then
4: |
5: [ProgressMessage].Text="No users in remote node!";
6: <- TRY_NEXT_NODE;
7: done;
8: |
9: [ListUsersObject].List = thelist;
10: [ProgressMessage].Text = "List of users in remote node is ready!";
11: <- CONNECT_NODE;
12: |
```

The first line is the header indicating that this is an event-handler for the message class LIST_USERS. The transferred data will be placed in a local buffer called thelist of type StringArrayType. It the first element of the received list indicates that no-users are present in the remote node, then at line-6 a message is sent to the application of class TRY_NEXT_NODE (no data are actually sent). The syntactic formalism for posting messages is very simple (notice that a variable must be explicitly used to send message - the contents of this variable are sent). Also, the syntax of the statements that cause events in shared space is provided.

```
MsgSend : <-to application Expression_msg class
|<-to application Expression_msg class Expression_data
|<-to lower-level Expression_msg class
|->to lower-level Expression_msg class Expression_data
;
CreateSharedObject : SHCREATE (Expression_exposed_data, IDENTIFIER_returned_id)
;
UpdateSharedObject : SHUPDATE (Expression_new_data, IDENTIFIER_passed_id)
;
DestroySharedObject : SHDELETE (IDENTIFIER_passed_id)
;
```
$Expression_{msg\ class}$ must be an integer value. In the formal rules which concern shared space, the translation schemes for $Expression$ must test whether the expression is being used within a communication-specific statement. In this case, a temporary variable will be created that will have type
the final type of the expression (according to type-conversion rules) and as content the evaluation of the expression. In case that Expression consists of only a single aggregate variable (such as an array or a record) that normally are not allowed within expressions, a copy of this aggregate structure is also made on a temporary variable (it is considered that the structure is used as a buffer). The temporary variable is used then as a buffer that is passed appropriately to the routines of the underlying communication package. Virtual interaction objects cannot neither be exposed in the shared-space nor be sent through the communication channel. An example of using these constructs follows:

Example-2

1:  sharedid opened_file_id;  
2:  sharedid equation_id;  
3:  string file;  
4:  ShCreate (a*x+b*y-3.14, equation_id);  
5:  ShUpdate (a*x+c, equation_id);  
6:  HandlerOn ShCreate (string, opened_file_id, file)  
7:  {  
8:    [Document].Text=file;  
9:    <- SEND_CONTENTS file  
10:   ShDelete(opened_file_id);  
11:  }

In lines 1,2 declaration of variables to be used as internal identifiers (can be also faced as pointers) to shared objects takes place. In line 4, a particular expression is exposed in shared-space. In line 5, the returned in 4 reference identifier is used to update the value of the expression. In 6, the specification of an event-handler concerning the instantiation of string objects is demonstrated. The variable file is copied with the contents of the shared string object. In line 10, the shared object (which may have been instantiated by another module) is destroyed, after requesting from the application (line 9) to send the contents of the file.
4.4.4. Virtual Object Genesis

Virtual Object Genesis concerns the introduction of new classes of interaction objects at an abstract level, without involving any type of lexically oriented concepts. Various classes of virtual interaction objects may be specified; for each such class a physical instantiation scheme must be specified at the lower-level.

Virtual objects may be implemented as either lightweight (i.e. internally not represented as dedicated structures) or heavyweight objects (i.e. internal representation through concrete and distinguishable structures), depending upon the adopted dialogue management strategy. In the proposed approach it has been considered more suitable to support heavyweight structures (in a heavyweight implementation, virtual interaction objects are explicitly represented internally as discrete structures, and the various notational conventions allowed in the language, are internally translated to equivalent constructs). The notational formalism has been designed so as to reflect the adopted abstraction scheme for interaction objects (see chapter-4.3.1.1.) that clearly manifests the practical ability to construct interactive entities which may be flexibly associated with various behavioral and morphological pictures. The following is the formal description of the syntax for object genesis:

VirtualObjGenesis : GENESIS EnvironmentSpec OF IDENTIFIER

[ ListOfMethods
  VariableDeclarations local data
  ConstructorSpec
]
;

EnvironmentSpec : VISUAL
| NONISUAL
| DUAL

ListofMethods : METHODS [ IdentList names of methods ]
;

ConstructorSpec : CONSTRUCTOR ( VariableDelcarations )
ConstrPassData only for composition
Compound
;

ConstPassedData : IDENTIFIER child object ( IdentList variables )
;

The non-terminal VariableDeclarations corresponds to the structural aspects of a virtual object that the designer has to specify according to the desired properties and its interactive role. These structures are notationally visible via the special operator [ ]. For example, [MyObject].Variable references the local variable Variable of the virtual object MyObject. Through this notation, the local variables of objects can be normally engaged in various statements (syntactically, the relation is similar to the record-field relation in Pascal). The notation will also be extended for composite objects. The constructor is a function to be called upon instantiation of this object at the virtual level. Various parameters may be passed to the constructor (it is a conventional function), and all the local variables of the virtual object are accessible within the constructor. It is also legal to define a recursive call of the constructor. ListOfMethods is necessary so that the parser can perform checking of whether a method class used by the designer is actually supported for a particular virtual interaction object.

<table>
<thead>
<tr>
<th>Higher Level</th>
<th>Virtual Instantiation (run-time)</th>
<th>Virtual Instantiation Logic (specification time)</th>
</tr>
</thead>
</table>

May 1993
Page 68
ICS-FORTH
**Lower Level**

| Physical Instantiation (run-time) | Physical Instantiation Logic (specification time) |

Providing notational access at the higher-level for data passed from the physical methods

Usually, a conventional callback routine (toolkit-oriented conception for methods) returns data specific to the interaction object, such as information related to the various external phenomena which triggered a callback or state description. Such information is likely to be necessary for method in order to decide which operation to carry out or what actions to take. Consequently, a scenario is required to allow such messages to be passed easily (and transparently to the designer) at the higher-level. The following is an algorithmic scheme which can be followed for virtual object genesis (the scheme extends also to instantiation logic which will be discussed later-on):

I. Introduce for each virtual interaction object special local variables, that will correspond to the interesting pieces of data returned from physical methods. At the lower level, for each interaction object, the following rule must be followed: for each distinct physical method, a set of additional attributes must be specified, which will be dedicated to storage of data passed from physical methods; the lexical technology server will be responsible to maintain appropriately and consistently these attributes, according to the status of the corresponding physical instances. This protocol is realised upon technology integration. Finally, upon description of instantiation rules for virtual interaction objects, an appropriate association between local variables which should carry method-specific data with such special physical attributes must be drawn.

**Associating physical attributes with (dynamically changing) variables owned by application / higher-level**

If such variables are included in the local variables of a particular interaction object, then the dependency may be normally specified via the physical instantiation logic. Otherwise, the association can be depicted via dependency schemes, which allow external shared variables to be associated with local variables. In this case, the dependency will be specified at the lower level, by setting the variables owned by either the application or the higher-level as external and shared.
Example-1

1: Genesis DUAL of Menu
2: [
3:    integer TotalOptions;
4:    integer* ApplMessages;
5:    integer SelectedOption;
6:    
7: Constructor (integer options; integer* msgs)
8:    [
9:        TotalOptions=options;
10:        ApplMessages=msgs;
11:        SelectedOption=-1;
12:    ]
13: ]
14: 
15: Method Select Menu
16: [
17:    <- ([Me].ApplMessages[[Me].SelectedOption]) [File].Text
18: ]
20: Menu commands(4, command_msgs);

Within lines 3-5 declaration of the local variables for Menu takes place. ApplMessages is declared to be of type integer* so that it can be used as an array of integers of varying length. The integer contents are actual messages classes to be sent to the application component (line 17), while the data accompanying a message are extracted from the local variable Text of the virtual interaction object instance File. Within the constructor, the local variables are initialised with the appropriate values. In line 19, initialisation of the array to hold the values for message numbers takes place, where in line 20, the instantiation of a sample virtual object is presented. The context in which such instantiation is legal is limited to the descriptonal scope of Dialogue Agents. Such a declaration may be specified in-between statements, in which case each time the control passes through that statement a virtual object instance will be created automatically. Additionally, its scope of applicability is bounded within constructors of Dialogue Agents. Next, the syntax for instantiation virtual objects (at the higher-level) is presented.

InstantiateObject : IDENTIFIER class IDENTIFIER instance
( ExpressionList constructor arguments ) ;
| IDENTIFIER class IDENTIFIER instance
( ExpressionList constructor arguments ) MethDefinition ;
;
MethDefinition : :: IDENTIFIER method class = IDENTIFIER method instance
;

Declaring objects must comply with all normal scope rules that hold for variables. It should be noted that because of the fact that through dynamic instantiation the virtual objects which are created may only be locally accessible within block of statements (outside of blocks they are not accessible), by the time of instantiation are methodless (no method is associated with them). For this reason, a special notation has been introduced so that a method can be easily assigned. This could be possible via a conventional assignment statement, however, in case of simple declarations (outside blocks of statements) it would not be possible to assign a method to a particular object without being in a block. The rule for MethDefinition may be extended if more attributes would be considered necessary at the virtual level. Also, upon instantiation of a virtual object within a Dialogue Agent constructor, the object is automatically considered as a member of the collection of objects controlled by that agent.
Fig. 4.4.4.1: Code Generation Scenario for Classes of Virtual Objects

Table of functions containing the code generation counterpart of every method found in dialogue specification.

Table of constructors functions for the various classes of virtual interactions objects.
4.4.5. Virtual Object Composition

Virtual object composition concerns the construction of synthetic interaction objects in the virtual domain. In the supported conceptual model, a fundamental difference between composite and primitive interaction objects is that while the later will correspond directly to particular physical interactive entities, the former will only provide an appropriate physical framework (which may not be necessarily realised by means of a specific physical interaction object) for component objects. According to this scheme, the constructions of various objects, such as dialogue boxes, is facilitated. Intuitively, composite objects could help in supporting generalised and reusable dialogue patterns.

- A composite object is defined as a flat collection (i.e. a set, not a hierarchy) of existing virtual objects, which may be either primitive or composite. Hierarchies are not explicitly supported (however, they may be specified indirectly, through composite children objects).

For each virtual composite object, an appropriate physical grouping specification is required (for each lexical environment) that describes the various lexical structural properties (topology) of the children in the perspective of physical unification. For instance, physical grouping could be expressed in terms of presentation-oriented object hierarchies.
Composite interaction objects are methodless. However, methods may be associated with the various children objects. All children objects are notationally visible through the special children resolution operator \( \_ \).

It is allowed the construction of dual virtual interaction objects, which may have children objects which are either only visual or non-visual. In this case, the dialogue control should take care so that at the lower-level only these objects that are meaningful (for the particular environment) will be instantiated. If a composite object is not dual, then it should be checked that all children objects are meaningful for the target specified environment. If so, all children objects are treated as if they have been related with that environment.

Example-1

For instance, consider the following virtual interaction objects:

```
dual Menu
visual Icon
dual TextEntry
nonvisual SpeechOut
dual Command
```

then a possible physical grouping description for each environment could be:

```
Non-Visual Grouping
   NonVisualWindow ---- |------ Menu
                      |------ TextEntry
                      |------ SpeechOut
                      |------ Command

Visual Grouping
   Form ----|----- Form ----|--- Menu
           | |--- TextEntry
           |----- ViewPort --- Form ----|--- Icon
                         |--- Command
```

```
VirtualObjComposition : COMPOSITION EnvironmentSpec OF IDENTIFIER
    [ ChildrenObjectsSpec
      VariableDeclarations
      ConstructorSpec
    ];

ChildrenObjectSpec : ChildrenObjectSpec InstantiateObject
    | InstantiateObject
    ;
```

It should be noted that no consideration is actually made with respect to the various physical aspects of the grouping method. The realisation of grouping via object hierarchies is only one possible interpretation that is supported under the proposed formalism. For instance, in a conventional X-based toolkit, grouping could be implemented through dedicated container widgets (such as Form or Box in Athena widget set). However, not only such a restricted type of spatial grouping in 2D space is possible. For example, consider that 3D display techniques are employed in the non-visual display environment with audio representations of particular physical interaction objects. Also, in existing visual techniques, the various objects are spatially distinguished (they own different portions of the
display surface) or overlapped (one over the other). In that case, a container object need only arrange appropriately children object through various layout structuring policies. However, if the different interaction objects are physically located by means of temporal sequencing (one object is displayed after the other - consider phonetic representation of menu options) the expressive facilities to describe topological properties are inappropriate.

Example-2

1: Composition DUAL of DialogueBox
2: [
3: TextEntry user_text;
4: Command the_ok;
5: Command the_cancel;
6: Command the_no;
7: Constructor (string initial_text; integer ok, cancel, no)
8: user_text(initial_text)
9: the_ok(ok)
10: the_no(no)
11: the_cancel(cancel)
12: [
13: ;
14: ]
15: ]
16: ]
17: DialogueBox confirm_delete;
18: Method Select [confirm_delete].the_ok
19: [<- DELETE [confirm_delete].user_text;
20: ]
21: ][confirm_delete].the_cancel]::Select = CancelFreeMethod;

Within the previous example, a lot of the notational and semantic features of virtual object composition are outlined. First, the technique of passing parameters to the constructor of children objects is demonstrated (lines 9-12). At lines 18-21 it is shown how methods for children objects can be specified. Specifically, considering that no facility is provided in order to assign, upon instantiation of a particular composite object, free methods to its children, an appropriate formal scenario is required to cope with this problem. At line 22, a simple solution to this difficulty is presented. Following are some additional formal rules that are required to complete the main syntactic aspects for interaction objects.

SetMethodStatement : VirtObject :: IDENTIFIER method class = IDENTIFIER free method

VirtObjectScheme-1 : [ IDENTIFIER primitive ] | [ VirtObjectComposite . IDENTIFIER ]

VirtualObjectContent : VirtObject . Variable

VirtObjectScheme-2 : [ HierarchyPath ]

HierarchyPath : HierarchyPath # IDENTIFIER | IDENTIFIER

// scheme-1 : [[obj3].obj2].obj1] provides access to obj1
// scheme-2 : [obj3#obj2#obj1] provides access to obj1
4.4.6. Dialogue Agents

Dialogue Agents may be faced as conceptual processes that are responsible for controlling various virtual interaction objects (i.e. managing instantiation, method call, dependencies, etc). The activation of a Dialogue Agent is either dependent on a triggering (enabling) condition (instantiation precondition) or specified via a dedicated statement (explicit instantiation); an agent may be viewed as a specialized type of object that is dedicated to the control of other objects. It is noticeable that according to the proposed scheme dynamic instantiations of virtual objects are efficiently supported, considering that controlled objects are instantiated each time a Dialogue Agent is activated (it is possible to have simultaneously multiple instantiations of the same Dialogue Agent pattern).

<table>
<thead>
<tr>
<th>Attributes of Dialogue Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>o Global Identifier</td>
</tr>
<tr>
<td>o Parent Agent (if any)</td>
</tr>
<tr>
<td>o Triggering Precondition</td>
</tr>
<tr>
<td>(if applicable)</td>
</tr>
<tr>
<td>o Disabling Precondition</td>
</tr>
<tr>
<td>o Formal Arguments (if applicable)</td>
</tr>
<tr>
<td>o Private Local Variables</td>
</tr>
<tr>
<td>o List of Controlled Virtual Objects</td>
</tr>
<tr>
<td>o Methods for local objects</td>
</tr>
<tr>
<td>o Locally Visible Dependencies</td>
</tr>
<tr>
<td>o Constructor</td>
</tr>
<tr>
<td>o Destructor</td>
</tr>
</tbody>
</table>

Triggering Preconditions

In various implementations, a triggering precondition is considered to be automatically the logical negation of a disabling precondition. However, it is possible to have a precondition that may become true at a certain point, however, its logical negation after becoming true is practically meaningless. For example, consider the following condition: `create if shcreate (<object>)`. Here, the Dialogue Agent will be instantiated each time a new shared object (as specified) is instantiated in shared-space. However, this expression is true only for a limited time interval. After time $T$ every instance in shared memory may be normally considered old. Intuitively, it is an incorrect behavior exactly after time $T$ to destroy the associated Dialogue Agent automatically because the condition would have returned false (expired). On the contrary, in such situations a Dialogue Agent must be explicitly destroyed through a dedicated statement.
**Fig. 4.4.6.1.** Functional Model of a Dialogue Agent

- **Top Level Control**
- **Dialogue Agent**
  - updates
  - requesting method execution
  - parent agent destroyed
  - (physically)
    - instantiate object
    - destroy object
    - modify object
    - notify for agent destruction
  - testing for enabling/disabling preconditions
  - resolving internal dependencies
  - access
  - control

---

*Application Exported Objects*

*Virtual Interaction Objects*
Emulating Event Handlers on Communication Events with Dialogue Agents

Generally speaking, it is possible to emulate event-handlers for communicating events with Dialogue Agents by allowing an empty list of controlled objects. Formally, the last argument declares that the introduction of event-handlers (on the presence of Dialogue Agents) does not necessarily contribute to the expressive power of the language. The following is a simple scheme to prove this argument:

<table>
<thead>
<tr>
<th>creation condition</th>
<th>on specific communication event</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlled objects</td>
<td>no objects controlled</td>
</tr>
<tr>
<td>constructor</td>
<td>put the code from the body of the event-handler in constructor and add at the end an explicit statement to destroy that agent</td>
</tr>
<tr>
<td>destructor</td>
<td>say &quot;good bye folks, I think I am gonna die&quot;</td>
</tr>
</tbody>
</table>

Fig.4.4.6.2. : Organising Hierarchies of Dialogue Agents

Organising Hierarchies of Dialogue Agents

Any Dialogue Agent may have multiple children Dialogue Agents. In this case, activation is either allowed if its enabling precondition holds (or an explicit call for instantiation is made) and also all of its predecessor Dialogue Agents are currently active or become active in the next dialogue management round. A Dialogue Agent must be destructed if the disabling condition becomes true, or if one of its predecessors is destructed in the next round. Clearly, it is possible to emulate hierarchical organizations with flat collections of agents, and simulate the additional effects on the destruction and creation of Dialogue Agents via special variables on the enabling conditions (the algorithmic solution is straightforward). Also, every Dialogue Agent has full access on the local data of its predecessors.

Multiple Concurrent Instantiations of the Same Dialogue Agent Pattern
Clearly, multiple instances of the same Dialogue Agent may coexist (either a precondition becomes true many times, before any deactivation takes place or multiple explicit activations are met). Care should be taken concerning the decision for instantiation of a Dialogue Agent with respect to its enabling precondition. If in a round a precondition is found to be enabled, then the corresponding Dialogue Agent is instantiated only if the precondition was disabled in the previous round. In other words, only transitions from a disabling precondition to an enabling precondition are of interest. Another problem is related to the disabling condition. If simultaneous instances depend on the same disabling condition coexist, then if disabling precondition becomes true, all the instances may have to be disabled, leading to a possibly undesired behavior. In this perspective, explicit deactivation is proved more safe, considering that only the specific instance is actually destroyed.

Allowing Functions Calls within the Expressions Engaged in Dependencies and Preconditions

One of the problems that arise when permitting functions calls as part of either dependencies or enabling / disabling preconditions of Dialogue Agents is to distinguish when a function must be re-evaluated. For instance, it may be possible that the result returned from a function is calculated based on global variables; consequently, the value returned from the function depends upon these particular global variables. Hence, each time one such variable is updated, the value of the function is updated, so the condition must be re-evaluated. However, it is not a trivial task to detect such indirect dependencies. Consider a more complicated (and usual) situation where one function is normally called within another function. Clearly, the complexity of maintaining and inquiring dependency graphs for such relations requires considerable computational power which may introduce performance problems at the run-time management module. For this reason, the following assumptions are made:

- If a function $F(x_1,\ldots,x_n)$ is engaged in a particular expression $E$ that is critical for run-time management, with the call $F(E_1,\ldots,E_n)$, then for each $E_i$ where $E_i$ is the actual expression for formal argument $x_i$, then the dependency holds for $E$ in the following manner:
  1. If $E_i$ is a variable $V$, then $E$ depends on $V$.
  2. If 1) does not hold, then $E$ inherits dependencies of $E_i$.

According to this hypothesis, in case of a function call only the direct dependencies associated with the actual arguments will be considered. However, this is a reasonable assumption, considering that modular programming suggests that functions should be implemented as black-boxes when faced as basic programming building-blocks. The only channel of communication to and from a function is via its formal arguments and returning value, respectively.

```plaintext
DialAgent : DIALOGUE AGENT IDENTIFIER name InAgent EnableStyle
[ VariableDeclarations ObjMethodDecls ConstraintSpecs
  CONSTRUCTOR Compound
  DESTRUCTOR Compound
]
;

InAgent : IDENTIFIER parent agent
| /* empty */
|
EnableStyle : ByCondition
| ThroughCall
|
ByCondition : CREATE IF ( BoolExpression )
DESTRUCT IF ( BoolExpression )
;

ThroughCall : ARGUMENTS ( VariableDeclarations_{formal args} )
;

ObjMethodDecls : ObjectMethodDecls ObjectMethodDecl
| ObjectMethodDecl
;

ObjMethodDecl : InstantiateObject
| Method
;

BoolExpression : FALSE
| TRUE
| ( MsgEvent )
| ( ShdestrEvent )
| ( ShcreateEvent )
| ( ShupdateEvent )
| Expression REL_OP Expression
| BoolExpression BOOL_OP BoolExpression
| NOT BoolExpression
| ( BoolExpression )
;

DisableDialogueAgent : KILL
;

From the definition of the non-terminal BoolExpression it is clear that notification for communication events can be expressed as a conventional boolean construct. If during the evaluation of a boolean expression such an event-based term is found, the communication space is queried appropriately for the presence of the specific event and in case of positive response the term returns TRUE, otherwise returns FALSE (also, the various parameters are set appropriately following the semantics of chapter-4.4.3.). However, the variables used in the event specification are declared automatically and are accessible within the body of the Dialogue Agent. The non-terminal InAgent provides the syntax for the specification of the parent Dialogue Agent (if any). Also, the two different methods of activating a Dialogue Agent are manifested by the two corresponding non-terminals ByCondition (declarative specification) and ThroughCall (explicit activation - procedural specification). Activation of a Dialogue Agent by-call is allowed even within constructors of Dialogue Agents.
Example-1

```
1: Dialogue Agent DisplayEmployee : CardOfEmployees
2: Create If (shcreate(EmployeeType, id, employee))
3: Destroy If (shdestroy (id))
4: [
5:   EmployeeDisplay emp_disp(employee.name, employee.position);
6:   Method Selected emp_disp
7:   [ <- SELECTED employee;
8:   ]
9: ]
10: Method Updated emp_disp
11: [
12:   shupdate(EmployeeType, id, employee);
13: ]
14: employee.name : [
15:   default : [empl_disp].Name
16: ];
17: employee.position : [
18:   default : [empl_disp].Position
19: ];
20: .
21: .
22: ]
```

In Example-1, it is demonstrated how the designer may specify that a Dialogue Agent should be instantiated each time an EmployeeType is instantiated in shared-space. The defined dependencies certify that the contents of the virtual object empl_disp are always the same with the local employee structure. Consider that the method update is called in case that the physical instance of the corresponding object is updated. Then, assuming that the correspondence with the employee structure is preserved, the contents are mapped to shared-space in order to reflect the recent changes. But this is possible only if the constraint would have already been satisfied prior to the call to Update method.

In order to ensure that at the statement of line 12, the employee structure is always consistent (at the time of call) with the most recent update of empl_disp and consequently the most recent value will be present in shared-space, the constraint satisfaction must always preceed the call of methods in the dialogue-control procedure.
Algorithm for Destruction of Dialogue Agents

Let \( U = \{X_1, \ldots, X_n\} \) be the updated objects in this round, considering that \( n \) is not 0.

Let \( D_i \) correspond to a Dialogue Agent description. Then assume the existence of two special attributes for this agent \( D_i.\text{will\_die} \) and \( D_i.\text{will\_born} \) taking boolean values. For a particular round \( r \), the first attribute denotes that all \( D_i \) instances will be destroyed in \( r+1 \), while the second attribute indicates that a \( D_i \) will be instantiated in \( r+1 \). Let \( D_i.\text{alive} \), taking boolean values, indicate whether \( D_i \) is alive in round \( r \) (a Dialogue Agent is alive at \( r \) if at least one instantiation exists at \( r \)).

Let \( D_1, \ldots, D_m \) be the set of Dialogue Agents that are active at this round (actually each \( D_i \) corresponds to a set of various possible instances for a Dialogue Agent description).

For each \( D_i \) where \( D_i.\text{alive}=\text{True} \) Do
\[
\begin{align*}
\text{If} & \quad \text{(disabling condition depends on an object } X_i \text{ in } U) \text{ Then} \\
& \quad \text{Re-evaluate disabling condition of } D_i. \\
& \quad \text{If} \quad \text{(disabling condition of } D_i \text{ is } \text{True} \text{) Then} \\
& \quad \quad \text{MarkChildrenToBeDestroyed}(D_i).
\end{align*}
\]

\text{MarkChildrenToBeDestroyed}(D_k)
\[
\begin{align*}
& \quad \text{If} \quad (D_k.\text{alive}=\text{False}) \text{ Then} \\
& \quad \quad \text{Return.} \\
& D_k.\text{alive}=\text{False.} \\
& D_k.\text{will\_die}=\text{True.} \\
& \text{For each direct child } D_j \text{ of } D_k \text{ Do} \\
& \quad \text{MarkChildrenToBeDestroyed}(D_j).
\end{align*}
\]

The algorithm ensures that when a Dialogue Agent is destroyed, then all of its children are also destroyed at the next round. When the function \text{MarkChildrenToBeDestroyed} reaches a dead child, then it knows that it is not possible to find a successor of this child that is alive (in order for an agent to be alive, all of its predecessors must be alive at that moment).

The previous algorithm deals with agents as classes (i.e. destroying based on disabling precondition), and does not cover the specific situation of destroying particular instances. Usually, the last case will concerns destruction by explicit statements. Also, such reorganization decisions will be committed massively at the end of the central dialogue control loop for more efficiency. For example, it may be decided that an object has been updated many times within only one round of control, however, only once at the end the object will be refreshed according to its most recent status. At the end of the control loop, the physical destruction of Dialogue Agents will take place.
Algorithm for Instantiation of Dialogue Agents

Let $U = \{X_1,...,X_n\}$ be the updated objects in this round, considering that $n$ is not 0.

Let $Q = \{D_1,...,D_k\}$ be the set of Dialogue Agents that will be destroyed in the next round (i.e. this is the output of the destruction algorithm).

For each $D_i$ where $D_i.alive=\text{False}$ and $D_i$ not in $Q$ Do

{ 
  If (enabling condition depends on an object $X_i$ in $U$) Then
  
  \{ 
    Re-evaluate enabling condition of $D_i$.
    \If (enabling condition of $D_i$ is True) Then
      \If (all predecessors of $D_i$ are either (alive and not in Q)) Then
        \{ 
          $D_i.alive=\text{True}$.
          $D_i.will_born=\text{True}$.
        \}
    \}
  \}
}

Testing that all predecessors of $D_i$ are alive includes the case where a particular predecessor is disabled in this round but is enabled on the next round. In order to ensure that this will also be satisfied the algorithm must guarantee that predecessors are first checked for activation prior to successors. In order to achieve this behavior, the algorithm must simply scan the Dialogue agents from up to down. However, this must be done pretty fast and, as will be explained later on, a special data-structure will be used to represent at the dialogue control the organization of Dialogue Agents.

In order to ensure that these algorithms will indeed work correctly and efficiently, an appropriate organisation of data-structures is required so that considerable performance can be achieved. For example, it is very expensive to have all the Dialogue Agent classes stored in a simple linear or non-linear (e.g. tree) structure where the all the nodes must be traversed, by testing whether an agent is alive or not. Certainly, the time is just proportional to the total number of Dialogue Agents, however, more efficient implementations are possible. One such implementation is indicated in Figure 4.4.6.3., where 4 different lists are updated in each round appropriately. For the destruction algorithm, only the AliveList may be traversed, while for the activation algorithm, only the DeadList should be checked. After the application of these algorithms the ToDieList and ToActivate list contain the new reorganization information.
Initially ActiveList is NULL, while DeadList contains all the agents, hierarchy by hierarchy, so that the members of a hierarchy are all together, and ordered in a breadth-first fashion. The algorithm for destruction ensures that DeadList and Alive List always follow this rule.

\[
\text{ActiveList} = \text{ActiveList} + \text{ToActivateList} - \text{ToDieList} \\
\text{DeadList} = \text{DeadList} - \text{ToActivateList} + \text{ToDieList}
\]

Fig. 4.4.6.3: Data Structures for Run-Time Management of Dialogue Agents
Algorithm for Computation Order of Updates

Constraint Resolution

Update N

D_d : Dialogue Agents for destruction
D_a : Dialogue Agents for activation

Fig.4.4.6.4. : Dialogue Control Scenario for Activation / Destruction of Dialogue Agents
4.4.7. Instantiation Logic for Virtual Interaction Objects

Instantiation logic concerns the set of rules describing the correspondence of virtual interaction objects to physical (toolkit specific) entities. Instantiation is defined by means of specifying dependencies between local data of virtual interaction objects, to lexical attributes. Dependencies are defined at the class level only, however, certain polymorphic instantiations (conditional instantiations) may be allowed due to the fact that multiple dependency schemes (with conditions that are checked at runtime, upon instantiation) can be described for a single virtual object. Next follows a preliminary formal scheme for specifying instantiation.

ObjectInstantiation : Instantiation of IDENTIFIER_class
[ InstantDeps
  AttrDefaults
  PhysicalGrouping
]
;

InstantDeps : InstantDeps InstantDep
| InstantDep
;

/*
We may have IDENTIFIER being either a local data object, or an attribute object. The former is the case when the Expression concerns data that have to be returned to the object, possibly due to method activation, while the later corresponds to situations where local data will be mapped through attributes of physical objects. */

InstantDep : IDENTIFIER_local data, attributes <-
    Expression_attributes, local data :
    BoolExpression_initial enabling condition ;
;

AttrDefaults : AttrDefaults AttrDefault
| AttrDefault
;

AttrDefault : IDENTIFIER_attribute = Expression_constant ;
;

From the formalisms that have been introduced, the first concept that will be discussed is that of AttrDefaults. An attribute identifier may appear in this rule only if it does not appear on the left side of any InstantDep rule. These attributes will normally be specified for each different instance of a class. However, using the AttrDefaults formal scheme, the designer is allowed to describe default values to be set each time an object is instantiated. In case that dependencies for attributes or fixed values have been specified for a particular object, the default value is ignored, and the specified attribute value is computed and passed to the lexical technology.
Algorithmic Scheme for Instantiation

Compilation: Upon compilation of the instantiation logic at the lower level and also of the virtual object genesis at the higher level, the following items must be generated:

a) A set of variables, each one corresponding to each local structure of the virtual object. Code for the declaration of these global variables will be generated for the lower level. Distinct identifiers will be used. These variables will be the place holders of the initial local data sent upon instantiation from the higher level to lower level.

b) Code for the evaluation of the dependency conditions. If local data of virtual objects are engaged, then these will be referenced using the global variables specified in a) for each class. The generated code will access such variables through the original identifiers.

c) For each dependency of an attribute to an expression, code will be generated to perform assignment of the expression to the specific attribute of the specific instance. For example, a statement like the following could be informally generated: `setAttribute (<Class>, <Instance>, <Attribute>, <Expression>).` It should be noted that upon instantiation, space is allocated to make a copy of attribute set for the newly created instance.

Run-Time: The following is a rough description of what will happen at run-time. Terms and algorithmic language are completely informal.

Step1: Receive message `[<INSTANTIATE>, <Class>, <Instance>, <[LocalData]>]. Extract local variables from `<[LocalData]> and assign values to the global set of variables of `<Class> (as previously specified).

Step2: Evaluate the conditions for each dependency scheme and mark the conditions that are true. This set of marked conditions will characterize the instantiation dependencies for this instance.

Step3: For each marked condition, execute the associated statement that satisfies the actual dependency (i.e. an indirect assignment must be performed generation of an appropriate function call). This statement will commit the assignment of values, either to the higher-level or to the lexical technology. It should be noted that a list of the objects involved (by general name) involved in the instantiation dependencies will be kept, so that each time an object is modified, the effect on the dependent objects can be propagated by the update propagation mechanism.

It is also important to decide what is the communication between the higher-level dialogue control module and the lower-level dialogue control concerning run-time instantiation of virtual interaction objects. Also the sequence of actions taken by the lower-level dialogue is important, in order to perform instantiation exactly according to designer’s specifications. Intuitively, a single message of the type `[<INSTANTIATE>, <ClassName>, <InstanceName>]` is not adequate to perform a physical instantiation, due to the fact that no information is actually known about the local data of the object, so that the instantiation agent can resolve completely the dependencies, decide the instantiation scheme for the object and finally compute and assign values accordingly. Following is an algorithmic scheme which will propose an initial tactic for instantiation at run-time, that will also depend on specific actions that should be taken upon compilation of the instantiation logic. Next follows the syntax for the specification of physical grouping.
PhysicalGrouping : VISUAL : PhysicalFramework;
    NONVISUAL : PhysicalFramework;
;
PhysicalFramework : IDENTIFIER basic grouping framework = Framework
    Framework,
    IDENTIFIER child of composite
;

Example-1

Visual Physical Grouping :
NonVisualWindow = [Menu, TextEntry, SpeechOut, Command];

Visual Physical Grouping :
Form = [Form [Menu, TextEntry], ViewPort [Form [Icon, Command]] ];
4.4.8. Main Dialogue Control Procedure for the Higher-Level

Start:
Consider all initialized variables as updated objects;

while (ForEver) do {
    Check for updates on shared variables;
    Check for update messages for virtual objects;
    if (objects involved in dependencies changed) then
        do constraint satisfaction;
    Check for message requesting method call for an object;
    if (method call has been requested) then {
        Set up global structures appropriately for relevant object;
        Pick-up appropriate corresponding function and call;
    }
    if (remaining updates from method) then
        do constraint satisfaction;
    Check for specific communicating events that have pre-specified event handlers;
    if (a communication event is pending) then {
        Set up local structures for storing incoming event appropriately;
        Pick up associated handler and call;
    }
    if (remaining updates from handler) then
        do constraint satisfaction;
    do destruction of dialogue agents;
    if (should destruct agents) then
        Commit destruction of these agents;
    do instantiation of dialogue agents;
    if (should instantiate agents) then
        Commit instantiation of these agents;
}

Note: The dialogue management procedure will terminate when an explicit TERMINATE statement is executed. The internal constructor / destructors of virtual objects and Dialogue Agents are responsible for sending appropriate messages to the top-level control.
5. Summary, Discussion and Conclusions

Currently applied solutions for the accessibility of GUIs by blind users, are not considered satisfactory and promising for the future; lexical adaptations and dialogue reproduction based on visual concepts, concerning specific characteristics of the general graphical environment, provide access methods which are too restrictive. We consider Dual User Interfaces as an appropriate methodological framework for developing User Interfaces, which takes into account the abilities and requirements of blind people. Issues related to the accessibility by blind users and interaction in a non-visual form, are explicitly addressed in the various design metrics of the proposed dialogue specification methodology, which is flexible enough not to restrict the dialogue for either the blind or sighted user. The design of the run-time architecture, mirrors the various involved levels of abstraction, when dealing with Dual dialogues. The technological basis of the proposed framework is entirely within the UIMS world. Existing UIMS models, such as Seeheim [HAGE90] or Arch / Slinky metamodel [UIMS92], and more implementation oriented architectures related to organisation of User Interface software, like PAC [COUT90], have been taken into consideration. It should be noted that our methodology for dialogue specification and run-time management of user-computer communication is based both on an introspective analysis (related to the internal aspects of dialogue control - esoteric dynamics) and morphoscopic analysis (concerned with the external morphological properties).

The proposed approach is characterised by a considerable flexibility at the lexical level. In principle, it is possible to utilise the facilities of an existing tool to enrich the original set of the physical interaction methods supported by the lexical technology, and consequently install newly generated methods in our architecture for further exploitation concerning the development of Dual User Interfaces. However, in order to facilitate incorporation of such additional libraries of interaction methods, this particular tool should provide the means of accessing efficiently the internal implementation modules. For instance, PERIDOT [MYER88] is a well known example of a UIMS enabling the development and combination of User Interface constructs by interactive demonstration; interactively developed constructs could be integrated in our framework by means of common dialogue techniques supplied through the visual lexical technology (only if PERIDOT generates distinct modules, accessed as conventional reusable code libraries). According to this paradigm, a meta-layer of dialogue management is actually introduced.

It is very important to allow flexible reusability of formal descriptions in terms of parameterised specification segments, in order to avoid "reinventing the wheel" when similar problems are actually faced. For this purpose, object oriented modelling is seen as an appropriate ground to address efficiently this issue. The basic concept is to provide a mechanism for enabling appropriate augmentation of the interaction methods supported by the lexical technology; facilities to allow intervention on these original techniques are not incorporated (i.e. to extend the set of available physical interaction objects). A similar approach is found in [CARE90] where the notion of Actors is introduced.

Additional functionality could be incorporated by means of handling user-oriented knowledge through formal methods of user modelling. Typical categories of user models include theoretical cognitive models, models of user knowledge, models of user characteristics, user task models and user views (reviewed in [SUTC88]). The application of formal description methods concerning user models aims to provide a machine representation of such knowledge for further automated analysis in order to achieve enhancement of a system’s interactive behavior. Specialists in human-engineering and human information processing may be involved in the detailed specification of the user models, for both
sighted and blind users, utilising appropriate tools provided within the interface development environment. Also, during user-computer interaction, it is possible to collect useful information through behavioral measurements \cite{MONK84}. It is a relatively simple matter to build monitoring functions in a prototype interactive system. Useful insights into the strengths and weaknesses of the dialogue can be often obtained by a methodological examination of recorded information after performing experimental interaction with the prototype system.

A linguistic specification method has been adopted, due to its appropriateness for non-presentation oriented formal descriptions; interactive graphical tools could be utilised only in terms of some additional functionality. These proposed methods may be efficiently employed to various heterogeneous graphical environments (i.e. X WINDOW SYSTEM and related toolkits, MS-WINDOWSTM and MACINTOSh\textsuperscript{TM} graphical environment, etc), due to the multiplatform nature of our architectural framework. The major aspects related to the design of various run-time management algorithms such as automatic update propagation, constrain satisfaction, dynamic or static object instantiation have been thoroughly addressed. Additionally, the construction of the proposed multilevel (linguistic) formal specification methodology takes under consideration particular problems related to the definition of appropriate parsing engines and code generation schemes.

Within the domain of interface development technology, UIMS based tools have been commercially established in the past, and it is expected that, in the near future, they will dominate the market. In accordance to the main objective of the TIDE-GUIB project, which is to ensure continued access for blind users on future computer-based applications, the technical definition of such an innovative proposed framework aims to efficiently and realistically enhance the restricted capabilities of existing tools in order to sufficiently overcome, at an appropriate early chronologically point, future accessibility problems. It is a fact that the various existing visual interaction techniques will be gradually substituted by more advanced highly interactive techniques employing 3D representations and virtual reality methods. Moreover, extensive use of visual symbolisms and visually oriented metaphors will be employed (e.g. color encoding, iconic representations, 3D effects, etc). It is evident that in this situation the provision of accessibility through adaptations at the lexical level is not practically feasible; a higher-level knowledge of the dialogue intrinsic properties is required so as to interpret and appropriately reproduce the various aspects of interaction that are meaningful for blind people. Apparently, such technical approach is practically possible only if higher-level description of the Human-Computer dialogue are supported. It has been considered one of the primary design goals to support an open architecture, which would be applicable to various platforms and graphical environments; this is because at present, there are numerous widely accepted environments with diverse abilities and differing interactive characteristics (lack of standardization). Considering that currently there are no implementational building-blocks concerning non-visual interaction, such an additional technical characteristic becomes practically a necessary prerequisite. It is believed that the proposed framework opens the road for integration of blind and sighted users, through the novel concept of Dual Interfaces, and provides a complete theoretical ground for further target implementations and experimentations.
References


[MONK84] MONK, A. Fundamentals of Human-Computer Interaction, ACADEMIC PRESS 1984


[MYER90] MYERS, B. A. A New Model for Handling Input. ACM Trans. Inform. Syst. 8, 3 (July 1990), 289-320


[SUTC88] SUTCHLIFE, A. Human-Computer Interface Design. MCMILLAN ED. LTD, 1988


