Interacting with the Disappearing Computer: Interaction Style, Design method, and Development Toolkit

Anthony Savidis, Constantine Stephanidis

ICS-FORTH Technical Report

TR317, December 2002

Abstract

In the context of disappearing computing, the user is engaged in mobile interaction sessions, with wearable machinery, while the software should be able to dynamically utilize distributed remote I/O resources, for the purpose of interaction, that are engaged (i.e. come and go) "on-the-fly". The technical purpose of this work is to provide the development infrastructure for crafting interfaces that support such disappearing computing behaviour. More specifically, the results of this work concern: (a) an interaction style that is particularly suited to the mobile, distributed and wearable nature of interactive applications; (b) a systematic design method, enabling designers to formulate easily dialogues with concrete interface elements, assuming dynamic engagement through discovery, and optimal utilisation of I/O resources "on-the-fly"; (c) an implementation library in the form of an interface toolkit, through which programmers can implement fully working interfaces, that hide all dynamic remote I/O resource management details; and (d) a set of run-time utility components, such as an application manager, being the necessary accompanying run-time instrumentation for dynamic distributed I/O control supporting multiple concurrent applications.

Table of contents

1. Introduction

Today, computing is facing a transition from the traditional desk-top to increasingly mobile and distributed interaction paradigms. In this context, concepts such as ubiquitous computing (Weiser, 1991), tangible bits (Ishi and Ullmer, 1997), nomadic interaction and migratory applications (Bharat and Cardelli, 1995), continuity of interaction (Massink and Faconti, 2001), and ambient intelligence (ERCIM, 2001) have emerged, to emphasize: (a) the technological shift from one single stationary computing platform to a networked world of distributed interactive computing devices, and (b) the challenge represented by the pursuit of high quality of interaction within those new paradigms, through the provision of appropriate methods, frameworks and tools both for the design and engineering processes.

Seen through the perspective of ubiquitous computing, the purpose of this work is to provide an infrastructure allowing users to engage in mobile interaction sessions within environments of varying computational (software and hardware) resources. Central to this work is the notion of a User Interface framework, comprised by:

- An interaction style, that is particularly suited to the mobile, distributed and wearable nature of interactive applications;

- A systematic design method, enabling designers to formulate easily dialogues with concrete interface elements, assuming dynamic engagement through discovery, and optimal utilisation of I/O resources “on-the-fly”;

- An implementation library in the form of an interface toolkit, through which programmers can implement fully working interfaces, that hide all dynamic remote I/O resource management details;

- Run-time utility components, such as an application manager, being the necessary accompanying run-time instrumentation for dynamic distributed I/O control supporting multiple concurrent applications.

In this context, the following key design requirements are considered:

- Mobile, wearable, wireless devices. Some, though not all, of these devices may host hardware used to facilitate any I/O between the user and an application (e.g., buttons, screens, microphones, speakers, etc). Such UI–specific I/O hardware is to be generally regarded as remote I/O resources. In fact, since the User Interaction Framework Design targets user-application interactions, the term I/O is taken to mean UI–specific I/O throughout the present document.

- Concurrent engagement of multiple I/O resources and interfaces with state-persistent re-activation.

- Ability to dynamically engage or disengage I/O resources as those become available or unavailable in mobile situations.

- High-quality interface design to ensure interaction continuity in dynamic I/O resource control, especially when such dynamic events occur in the middle of interaction sessions.

It is worth mentioning that the Human-Computer interaction paradigm set forth deviates from its desktop analogue in two main ways:

- The I/O resources available for user interaction are distributed in the form of devices that communicate wirelessly.

- This computational environment is highly "transient" as devices join / leave a Body Network through borrowing / lending, come within / without range, power up / down, etc.

2. Related work

The work presented in this technical report relates to many domains, such as ubiquitous computing, nomadic interfaces and migratory interfaces. Comparison may concern various outcomes of the reported work, and in particular the overall application set-up, the interaction design style, the dialogue design method, and the software engineering strategy.

2.1 Continuity

The need to preserve the uninterrupted sequence in the bi-directional flow of communication between users and interactive applications has been identified first in (Norman, 1990). The notion of continuity has been given particular attention under the EU TMR TACIT Project (see Acknowledgements), which has emphasized the need for "an intensive exchange of constant updates of continuously changing information between human and system over an extended period of time" (Massink and Faconti, 2001).

An example of continuity in a real-life activity is discussed in (May, Buehner and Duke, 2001), where driving is briefly analysed as a situation in which distributed dynamic feedback from the environment occurs, requiring concurrent actions and continuous interpretation on behalf of the user. When the user is engaged in a mobile interactive session, in which the physical interaction environment changes (i.e. the I/O devices change), the interface should preserve continuity, even though it changes its interaction style and form. This requirement is discussed under the concept of plasticity in (Calvary et al., 2001), emphasizing the need for distributed applications to "withstand variations of context of use while preserving usability". The nomadic application development that is discussed in this paper will elaborate on the specific design and engineering approach that has been employed to accommodate for continuity and plasticity in dynamically changing contexts emerging in mobile interaction sessions.

2.2 Ubiquitous computing

The domain of ubiquitous computing has been introduced in (Weiser, 1991), envisioning an environment populated by computational resources in which users are provided with information and services at any time. In (Abowd and Mynatt, 2000), three key challenges are identified in the context of ubiquitous computing research, namely: natural interfaces (rich variety of communication), context-awareness (sensing information from the physical and computational environment), and capture and access of live experiences. It is argued that, although this represents taxonomy of work previously done in this domain, it does not account for all the various issues that need to be addressed in the context of ubiquitous computing. For instance, the environment context cannot be separated from the interface context, since in the specific case the *interface is the environment*. In this reality, the interface requires a kind of dynamic introspection as to what types of I/O resources are available at a given in point time, and at a particular location.

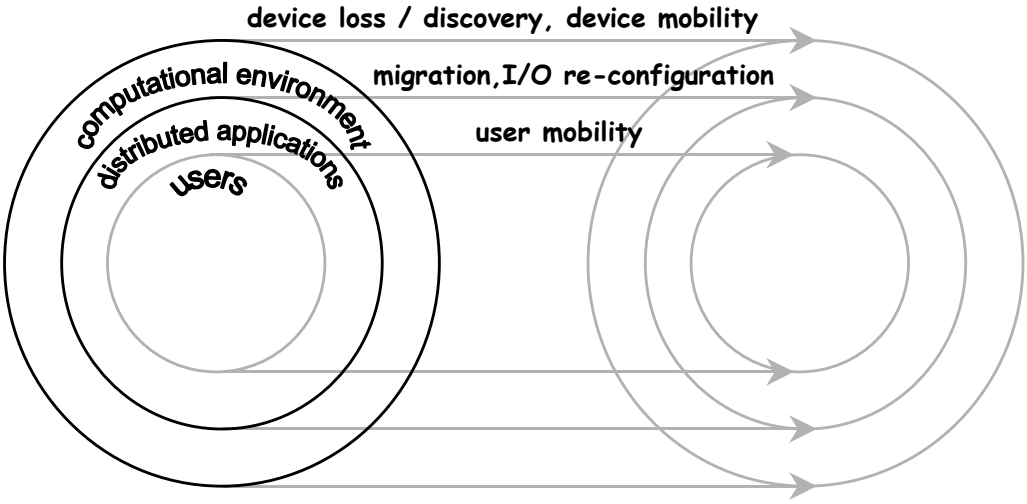

Figure 1: Ubiquitous computing viewed as mobile interaction sessions in dynamically changing environments, with nomadic applications that continously adapt to utilise the resources available.

Additionally, applications are not bound to a particular physical location or context, but may migrate and follow the user in mobile interaction sessions (see Figure 1). Consequently, additional issues that need to be addressed concern the dynamic update of the interface to accommodate for the changing I/O resources, the dynamic detection of resource updates, and the continuous (i.e., un-interrupted) migration while the user is moving. There has been prototype work regarding the engineering of applications featuring environment context awareness, based on the development of re-usable higher-level elements, similar to widgets (Salber et al., 1999). Those elements, though called "widgets" in (Salber et al., 1999), do not implement distributed / remote interaction with the user, but simply notify the run-time application of various types of events occurring in the environment, such as particular activities, or the presence of persons.

2.3 Migratory interfaces

The notion of migratory applications has been introduced in (Bharat and Cardelli, 1995), presenting the implementation of an application that could migrate from machine to machine, carrying state information so that it could re-activate at the exact point it left off. The original implementation has been based on the Obliq distributed scripting language (Cardelli, 1995), resembling Java, with built-in features for object distribution. The approach implemented in (Bharat and Cardelli, 1995) relies upon the implementation of a generic migration layer that can be used by developers without additional effort. The approach adopted in the work presented here is different, as: (a) it relies on the Java language capabilities to serialize both program code and data, and transmit them easily over the network; (b) it requires that all applications implement three special purpose abstract Java interfaces (corresponding to interface state, core state and activation behaviour); and (c) applications do not re-start exactly as they were at the point of suspension, since, from the new device, some I/O resources may not be network reachable, necessitating the explicit re-configuration of I/O prior to dialogue resumption.

2.4 Logical and abstract I/O behaviours

In this type of work, particular emphasis is put on capturing the generic input-specific behavioural aspects of interaction, in the form of generic highly parameterized primitives

through which complex behaviours can be assembled in a modular manner. Early systematic work in this field mainly concerns *interaction tasks* (Foley et al., 1984), providing a comprehensive theoretical framework, while *interactors* (Myers, 1990) provide an implementation-based framework for combining common input behaviours met in graphical interfaces for creating more complex interfaces. The concept of meta-widgets is more recent and represents the abstraction of interaction objects above physical platforms (Wise et al., 1995). A meta-widget is free of physical attributes and presentation issues and is potentially capable to model interaction objects above the level of metaphors. More implementation-oriented work with full-fledge support for the construction of abstract interaction object classes in a language-based UIMS has lead to the notion of virtual objects supported by the HOMER UIMS (Savidis & Stephanidis, 1995) and its successor the I-GET UIMS (Savidis & Stephanidis, 2001a).

When designing applications that provide interfaces with dynamically engaged I/O resources, the employment of logical I/O objects enables the design of dialogues without directly dealing with physical I/O details. At a later design stage, each logical object is associated with alternative instantiations, utilizing different instances of I/O resources. This type of separation between the logical and the physical layers is directly reflected in the interface implementation, making the overall engineering process far easier, and enabling modular extensions for different types of I/O devices.

3. The NOMADIC MUSIC BOX Application

During the very early stages of work in the development of an architecture for interface distribution, migration and dynamic I/O resource control, the need for a concrete application scenario that would serve both as a design and an engineering test-bed, became evident. In this context, a small application scenario has been purposely specified.

3.1 Application description and logical components

The initial application scenario, which has been selected, is a nomadic MP3 player, called the *music box* application. This application provides the basic functionality of an MP3 player (see Figure 2), i.e.:

- Downloading of MP3 files, on the presence of network connection, from a specific server address, where the user MP3 library resides.

- Local storage of MP3 files, on the presence of either a local file system or local data caching / storage. File management properties such as the number of files that can be stored or the capability to reference stored files through a descriptor (such as a file path) depend on the properties of the host computing platform.

- Decoding and play-back functionality for MP3 files through a software / hardware player, providing typical audio control functionality (e.g., play, pause, stop, next / previous file, etc.).

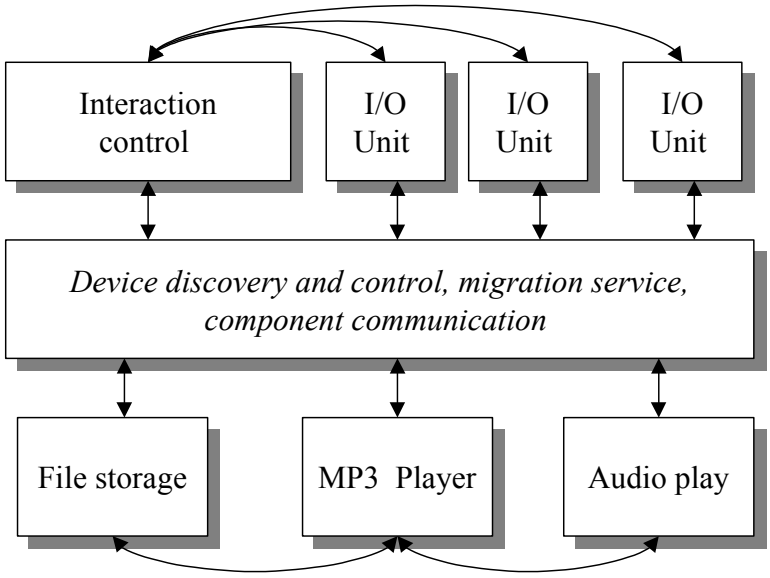

Figure 2: Logical components of the Music Box application**.**

3.2 Application Set-up and Physical Device Components

For the purposes of the application experiment a specific set-up of the interaction environment has been selected, as a collection of independent mobile devices with wireless connection. The selected devices fall mainly in the domain of consumer electronics, currently not offering the functionality that is required to directly employ them as open computing platforms. In the context of a "ubiquitous computing" installation, however, they are considered as typical examples of devices which in future may provide embedded operating systems and networking support. This scenario has been based on the definitions found in (Norman, 1998), where such devices are reflected in the notion of an information appliance, combining concepts from both graphical dialogues and consumer electronics. More specifically, the device units selected for the music box experiment are (see Figure 3):

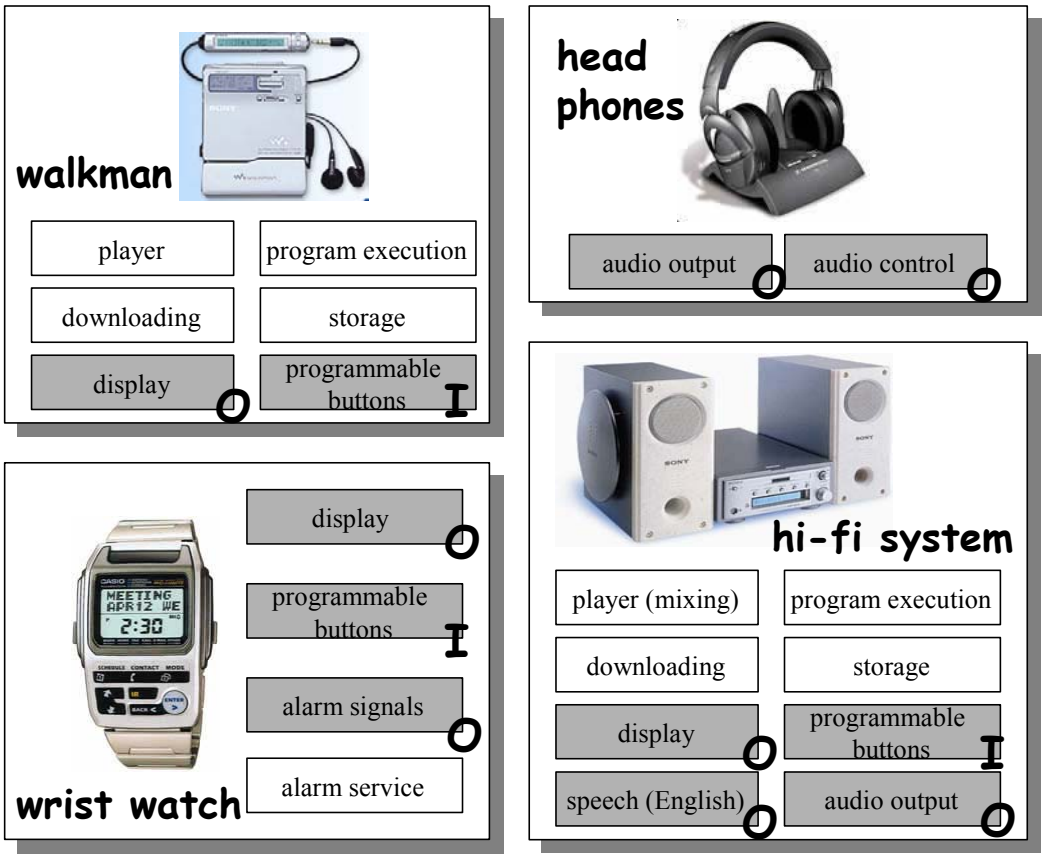

Figure 3: Device configuration and functional capabilities of the distributed music box application experiment; shaded rectangles represent interface-specific programmable functionality (O is for output, and I is for input), while white rectangles represent non-interactive services.

- *Portable music player unit* (e.g., walkman). It provides a hardware MP3 player (that can be turned off), local operating system with temporary file storage (main memory) and primitive program execution control, networking for downloading, hardware buttons for playback control with software assignment and client notification support, and a small LCD display (2 lines, hardware font, with 16 characters each) with a software API.

- *Head phones*. They provide audio output capability with wireless single-partner exclusive connection, accepting streamed audio data and audio control commands from a server whose address can be dynamically selected.

- *Hi-fi system*. This provides the functionality of the portable music player, supporting automatically (i.e., without the need for audio re-direction) audio play, and allows exclusive external use of the loudspeakers as an audio output service. It offers audio playback with mixing capability (multiple MP3 files playing concurrently) enabling audio redirection to an audio service, and supports speech synthesis for the English language. Also, it provides a software controlled character-based display (3 lines with 32 characters per line), supporting three built-in fonts, and display of 64 types of predefined icons (displayed from hardware at specific display locations).

- *Wrist watch*. It provides a software controlled LCD display, with one line of 16 characters, supporting 2 icons at fixed positions (beginning and end of line) from a collection of 8 icons that can be handled remotely via the BlueTooth™ protocol. Additionally, it supports the generation of 16 different tones (from a hardware defined library) that can be activated through software control, and provides two programmable buttons. Finally, it offers an alert service with two modes (count-down and absolute-time), providing a notification message to the client and, optionally, an alert tone to be played.

3.3 Dialogue design requirements

During the interface design of such a distributed application, in which the I/O resources were to be dynamically discovered, selected or altered, possibly for the same user task and in various situations in-between tasks, the following design requirements emerged, which provided an insight regarding the necessary methods and instruments for the conduct of the overall interaction design process:

- Due to the dynamic nature of I/O, it has been necessary to employ generic I/O modelling, introducing within the dialogue design multi-modal logical I/O, while all the various physical I/O resources had to be linked explicitly with their respective logical I/O category. Based on the terminology of (Foley et al, 1984), such logical I/O categories are called *I/O tasks*, while the concrete physical devices are called *I/O resources*.

- During interaction, for each I/O task, multiple available I/O resources may be detected at a particular point in time. In order to ensure that the most suitable I/O resource is always selected at run-time, the relative ranking of these resources during the design process has been necessary. If the concurrent employment of a set of I/O resources is required during interaction (i.e., multi-modal I/O), this had to be explicitly documented by assigning to the involved resource equivalent "selection scores". This type of design logic has been directly embedded within the target implementation, thus constituting part of the dynamic I/O resource control logic.

- During interaction, there are situations in which for any *active* I/O task (i.e., an I/O task that is required for a user-task which has been initiated at run-time, but not yet completed), the associated I/O resource may become unavailable due to loss of connection. In this case, an available alternative I/O resource for the specific I/O task

has to be detected and employed. However, if all the candidate I/O resources are used by currently active I/O tasks, the interface may have to make some dynamic re-assignments based on task criticality and alternative I/O resource availability. This type of design logic has to be appropriately documented and subsequently embedded in an implementation form within the run-time interaction control component.

- When interacting with applications in a distributed computing environment, it is critical that the user is enabled to: (a) manage alternative applications and be able to switch among them; (b) choose the I/O re-configuration to be applied in case of dynamic device engagement, or loss of device connection; and (c) decide the migration and re-activation of the system to another "processing device unit". This issue raised the need for designing the user interface of distributed applications to provide three categories of functionality (see Figure 4): application-specific dialogue, application management, and migration / dynamic I/O resource control.

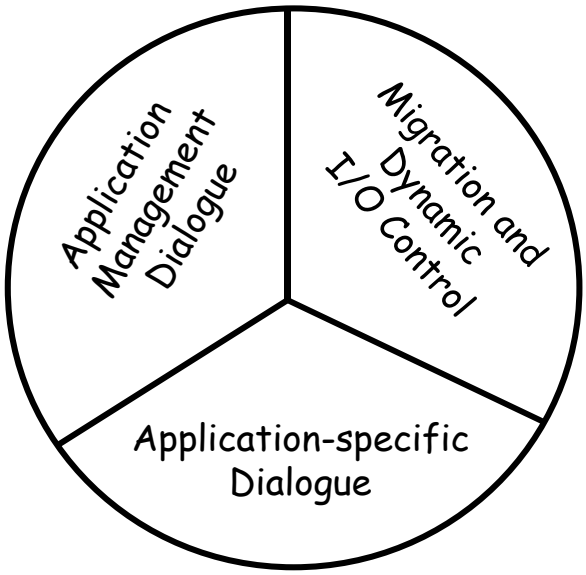

Figure 4: User interface design categories for distributed nomadic applications.

3.4 Interface design process

The interface design process has been carried out in strict cooperation with the lower-level system set-up and the application prototype-development process. Due to the highly experimental nature of the overall application scenario, it was naturally expected that some of the initial design ideas would not work in real practice, either because it would be practically proved that the expected interface quality could not be accomplished, or due to technical and implementation-oriented barriers. The main challenge of the interface design phase has been to ensure that, in such a dynamically changing computing environment, where devices are either dynamically discovered and used, or become unavailable and subsequently disengaged, the user should experience uninterrupted interaction, by aiming to constantly preserve the principle of dialogue continuity. This is a typical target in ubiquitous computing (Abowd and Mynatt, 2000), identified as the need for a continuously present interface.

During the early small-scale evaluation cycles, four categories of key usability issues have been identified (see Figure 5); the end-users have reported that those problems are mainly due to: (a) some initial confusion caused by the dynamic change of I/O resources while moving within the environment, as well as by the distributed nature of input and feedback; and (b) the physical diversity regarding the various displays and hardware controls available on the various I/O devices.

Usability issue	***Design issue***
Where am I?	*Context clarity*
What is this?	*Feedback interpretation*
What can I do?	*User tasks, I/O tasks*
How can I do it?	*Action sequences, I/O resources*

Figure 5: Key usability issues identified during the early small-scale evaluation cycles of the nomadic music-box application prototype.

Properties	**Input actions**	**Dialogue specification**
• List of options (strings) • Current option (focus)	• Next • Previous • Select	see Figure 7
Notifications	**Output actions**	**Dialogue instantiation**
• *Select*, supplying the current option (index and string).	• Set focus option (text content) • Select feedback	see Figure 8 and Figure 9

Figure 6: Dialogue design of the Selector I/O task with alternative I/O configurations.

In Figure 6, the design logic of a specific I/O task (the *Selector* task) is shown, demonstrating the pluralism of alternative I/O configurations that need to be considered during the design process. In the design documentation, each I/O task is broken down into logical *input / output actions*, while the dialogue specification for I/O tasks is typically provided in the form of a *state diagram.* This approach combines techniques such as interaction tasks (Foley et al., 1984) and interactors (Myers, 1990; Duke et al, 1994) with state-based dialogue specification (Jacob, 1988; Wellner, 1990) and polymorphic design instantiation of interaction objects (Savidis and Stephanidis, 1995).

The successive step of the design process is the *dialogue instantiation* phase, where the binding of the input / output action set to various alternative configurations of I/O resources is defined. Alternative configurations are ranked according to their selection priority. Following this approach, an I/O task can be sufficiently supported during interaction if there exists at least one configuration, within the I/O task dialogue instantiation logic, in which all

the necessary I/O resources are available. For the purposes of the application scenario four basic I/O tasks have been employed, which constitue a sub-set of the basic interaction tasks defined in (Foley et al., 1984): *selector* (single choice), *command* (performing a direct action, such as a button press), *numeric* (defining a numeric value, named as valuator in the context of interaction tasks) and *text* (for textual content).

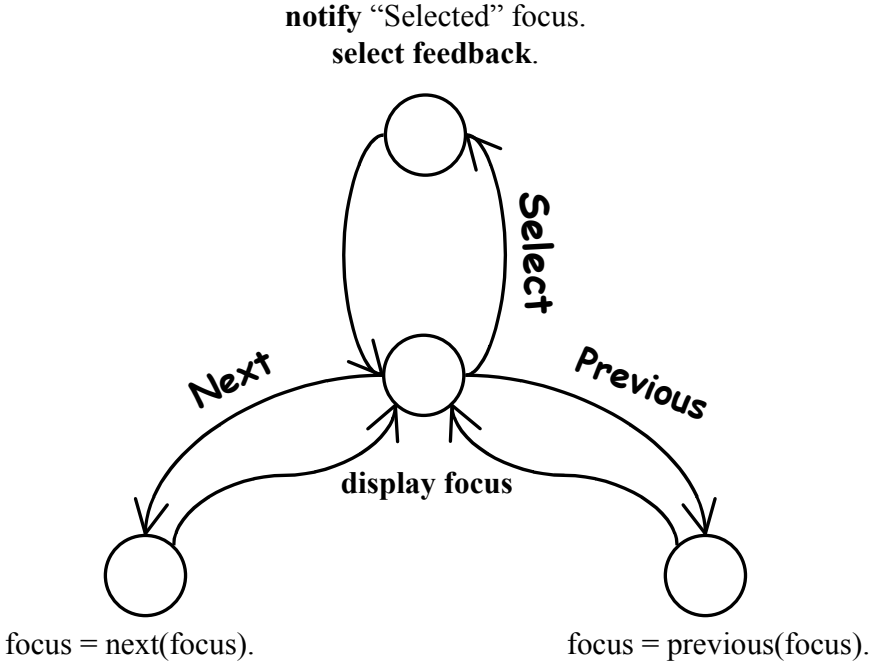

Figure 7: Sample dialogue specification of the I/O Selector task in the form of a dialogue state automaton (start state is in the middle). At each state, the action(-s) to be performed are outlined (in this example, bolded items represent output actions).

On the basis of these four I/O tasks, the three dialogue categories (i.e., application management, migration and I/O control, music-box control) have been appropriately decomposed. Following the design approach illustrated in the Figures 6, 7, 8 and 9, each of the four I/O tasks has been instantiated with appropriate alternative I/O configurations. As part of the discussion regarding the implementation architecture, it will be shown that each of the two design layers for I/O tasks (i.e., logical dialogue control – automaton, device level control – instantiation) has been mapped to a corresponding implementation layer. Subsequently, the dialogue structure will be briefly discussed.

Instantiation, *Selector*, with ranking 1.		
Requires	Input	• One (1) *programmable button*, ALWAYS.
	Output	• One (1) *text display*, single line, more than 16 characters, ALWAYS. • One (1) *audio signal* (programmable or hard-coded), OPTIONAL.
	Functional	• One (1) *time alarm* with countdown service, ALWAYS.
Implements	Input	• Select is performed after receiving a *press* notification from the programmable button. • Next is performed after receiving a "next" message from the time alarm (see Functional below). • Previous is not supported.
	Output	• Display focus is implemented by setting the displayed text to be the focus option of the Selector. • Select feedback is implemented by generating the audio signal.
	Functional	• A count down request is made, requiring a notification every second, that will return a "next" message.

Figure 8: Dialogue instantiation of the Selector I/O task, requiring the presence of one programmable button, textual display and time alarm service.

Instantiation, *Selector*, with ranking 2.		
Requires	**Input**	• Two (2) *programmable button*, ALWAYS.
	Output	• One (1) *text display*, single line, more than 16 characters, ALWAYS. • One (1) *audio signal* (programmable or hard-coded), OPTIONAL.
	Functional	• None.
Implements	**Input**	• **Select** is performed after receiving a *press* notification from the 1st programmable button. • **Next** is performed after receiving a *press* notification from the 2nd programmable button.
	Output	• **Display focus** is implemented by setting the displayed text to be the focus option of the Selector. • **Select feedback** is implemented by generating the audio signal.
	Functional	• None

Figure 9: Dialogue instantiation of the Selector I/O task, requiring the presence of two programmable buttons and a textual display.

3.4.1 Application management dialogue

The application manager is a separate application, that is aware of: (a) the available applications that the user may initiate (i.e., application registry): and (b) the applications currently running (i.e., process registry). In the adopted application scenario, aiming to simplify the target implementation, it was decided that during run-time, the user is allowed to interact only with the focus application. The application management dialogue supports three basic activities: switching to a running application, initiating an application and closing an application. As it will be shown, the overall dialogue structure is menu based. Typically, the last option of each menu presented is a "previous menu" option, closing the current menu and returning to the previous one.

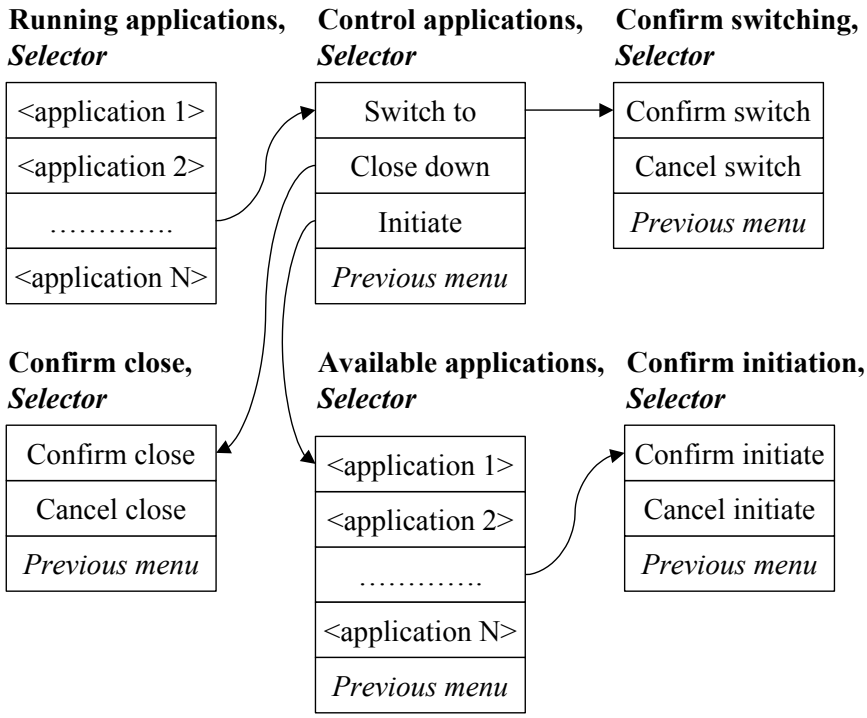

Figure 10: Selector hierarchy for the design of the distributed application management dialogue.

In the dialogue design below (see Figure 10 for an overview of the distributed application manager dialogue design), this option is omitted for clarity:

- *Running applications.* Displays the titles of all running applications (with the exception of the application manager). On the selection of an application, another menu is opened, presenting three options:

- *Switch to.* The dialogue focus is given to this specific application (at run-time, giving the dialogue focus to an application means distributing to the I/O resources used by this application a specific "focus gain" message).

- *Close down*. A confirmation menu (OK, Cancel) is initiated. If OK is pressed, the application is closed, and is also removed from the menu. If Cancel is pressed, the previous menu is displayed.

- *Initiate*. The list of available applications (excluding those currently running) is presented. When one option is selected, a confirmation menu is displayed (selecting OK activates the application and adds it in the list of running applications). In contrast to desktop computing, when an application is initiated *it does not* gain the focus, unless explicitly selected by the user.

3.4.2 Migration and dynamic I/O control dialogue

The migration dialogue is user-initiated and, as opposed to the application management dialogue, is embedded in the application-specific dialogue implementation. When the user selects the migration dialogue, the system requests through a dynamic registry the list of network reachable devices that are capable of hosting the application execution. The user may select to migrate the running application to one of the target machines (see Figure 11), and appropriate feedback regarding the result of the migration operation is provided. Migration is an ingredient which at the level of the application scenario cannot be effectively addressed. This approach is to be severely revisited after the first version of the implementation toolkit is completed.

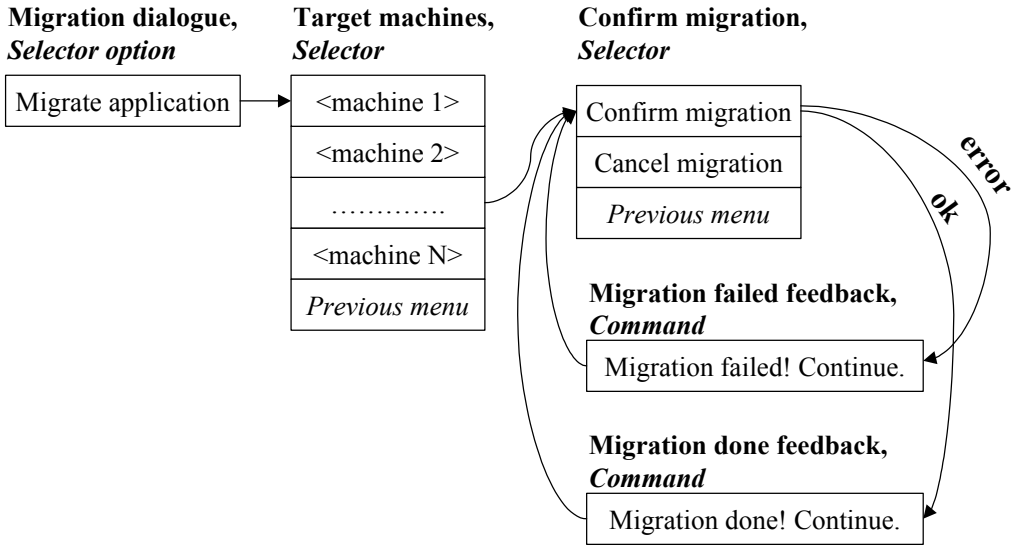

Figure 11: Selector hierarchy for the design of the distributed application migration dialogue.

While the migration dialogue is user-initiated, interaction regarding the dynamic detection of an I/O resource, or the loss of connection with an already used I/O resource, is characterized by its asynchronous system-initiated nature. When a utilised I/O resource is lost, an appropriate message is sent to all applications sharing this resource. If the receiving application owns the focus, it will directly inform the user (see Figure 12 direct feedback). Otherwise, applications will appropriately inform the user (see Figure 12, phase I) when they

regain the dialogue focus (see Figure 12, deferred feedback – phase II). The same holds in the case of dynamic I/O resource discovery. It should be noted that the feedback dialogue itself requires concrete I/O resources, which need to be allocated dynamically (i.e., a Command dialogue instantiation, see Figure 12).

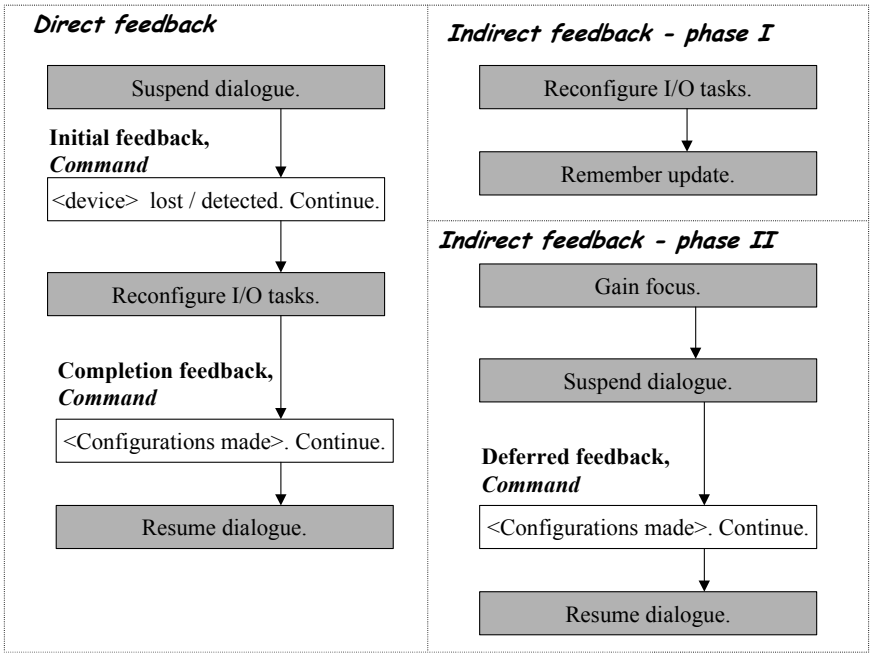

Figure 12: Dialogue for dynamic loss / detection of I/O resources. Shaded rectangles correspond to non interactive internal processing.

3.4.3 Music Box dialogue scenario

The music-box application dialogue offers three main categories of interactive functionality: (a) typical music control commands, offered by the MP3 player (either a hardware or a software one); (b) facilities to load from local storage or from network an MP3 file list, and select a particular MP3 file to play; and (c) linkage with the application manager and the migration control. The overall design structure of the music box dialogue is provided in Figure 13. For the purposes of the presented developments, when the local file list is selected, the MP3 file list is expected to be located in a specific local directory, at the local CD ROM drive. In the absence of a CD ROM drive in the current device, other available logical drives are checked in their OS specific order (e.g., C, D, E, etc).

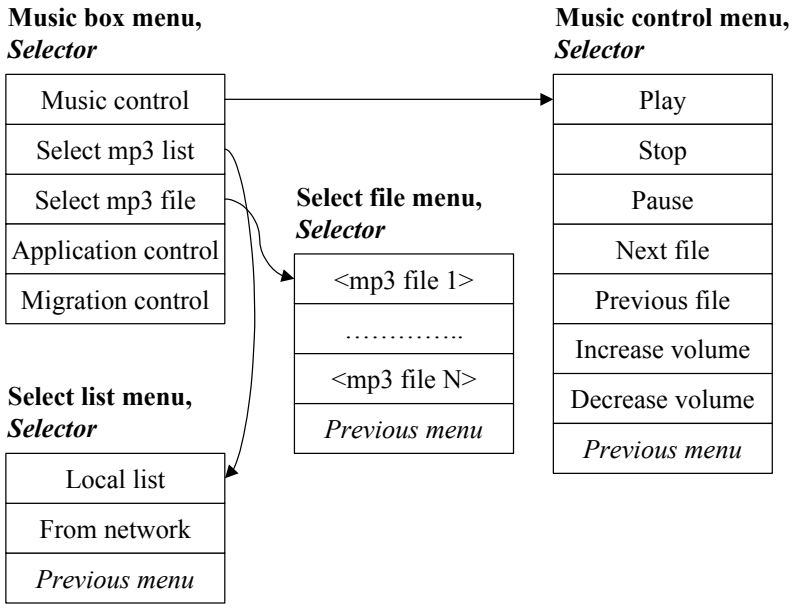

Figure 13: Menu hierarchy for the design of the distributed application music-box

When a network supplied list is requested, the music box application requests an "MP3 file service" from the available run-time components playing the role of the "service directory" (see Section on the application architecture). On the availability of such a service (more than one instances may be available), the music box requests the file list, and retrieves it locally. After downloading is completed, the application disconnects from this service.

4. Architectural Requirements

The development of such a distributed, migrating application, characterized by dynamic I/O configuration, requires an appropriate software architecture. In this context, the top level functional requirements of the target architecture are:

- To support state persistence, migration and re-activation.
- To enable effective and efficient expansion of I/O tasks with the implementation of newly designed dialogue instantiations.
- To enable the dynamic detection of I/O resources, functional services, and available applications.

Following these requirements, the first step in defining an appropriate architecture has been the identification of the distributed components, their respective registries (for lookup / discovery), and the run-time regulations (where it is run, when it is initiated and by whom, and how it is controlled). In this context, a set of run-time components has been selected (see Figure 14) for run-time use by distributed applications through special purpose protocols, implemented in the form of customized APIs. These components are:

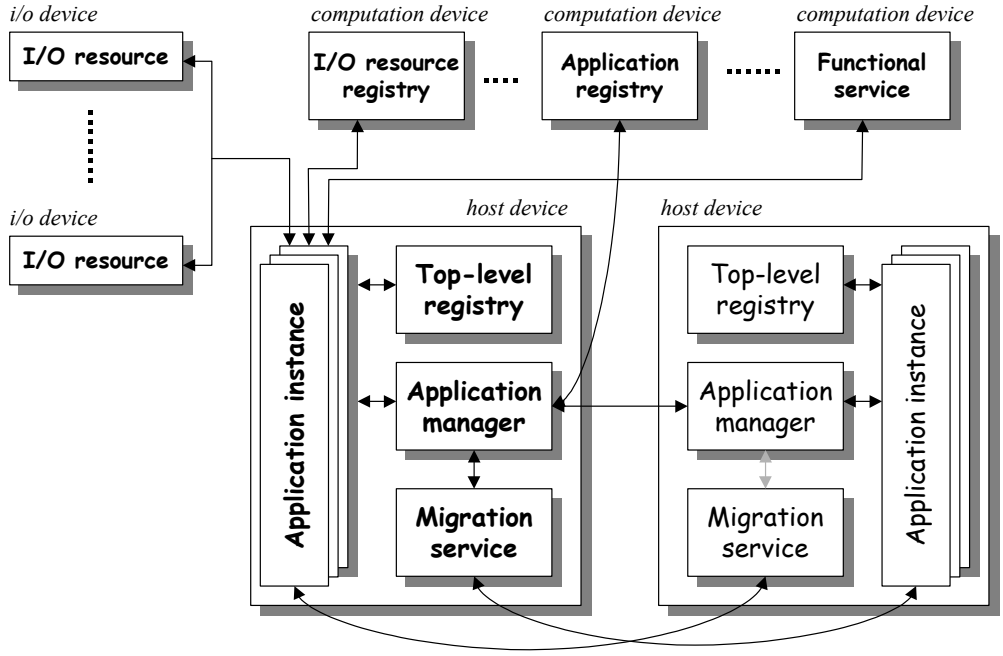

Figure 14: Components categories, device categories, and communication links in the distributed application *macro* architecture.

- *Top-level Service Registry*. This provides the linkage with all the various categories of services that are made available within the distributed environment (either made available interactively to the user, or internally employed by the running applications).

- *I/O Resource Service Registry.* This provides a look-up service for the dynamic discovery of I/O resources. Distributed applications need this registry to detect the available I/O resource services.

- *I/O Resource Services.* Those concern the available I/O resources in the form of distinct run-time service components.

- *Functional Services.* Those correspond to non-interactive services, such as particular computation / processing (like the MP3 encoder), storage, download, etc.

- *Migration and Initiation Service.* Provides support for migration and re-activation. When an application needs to migrate to a particular device, it will have to communicate with the migration service available on that device, through the migration API.

- *Application Registry.* This service collects the various application classes which are available and can be initiated by the user.

- *Application Manager.* The application manager runs on each device that is capable of running applications, and supports the initiation of an application locally. When the user requests the initiation of an application found in the application registry, the registry communicates with the local application manager to start-up the application locally. Additionally, the application manager maintains the list of running applications.

Applications may only run on devices that provide a top-level registry, a migration service, and an application manager. In the adopted architecture, these devices are called *host devices* and are to be distinguished from *I/O devices,* which provide I/O resource services, and devices providing functional services, resource registry, and application registry, which are called *computation devices*. A device may be simultaneously a host device, an I/O device and a computation device. Following this definition of device roles, in order to support a distributed interactive application, the presence of at least one host device, one input device and one output device is necessary (see Figure 14 for an outline of the various components and device categories, together with their communication links). Then, to enable migration, alternative host devices must be detected (through look-up of migration service components). Dynamic interface re-configuration is initiated (by communicating with I/O resource registries) when new I/O devices have been discovered, or the connection with some device is lost. The details of the migration and dynamic I/O re-configuration procedures are provided below.

4.1 Migration procedure

The first step in migration is the detection of a remote host device. The local application manager initially requests from the top-level registry notification of the presence of a remote host device. When such a notification is received, the local migration service of that particular host device contacted, to carry on with the migration process. When an application manager detects that communication with a host device is lost, it initiates a communication verification cycle with running applications (based on time outs), and removes from the list those that fail to communicate in the short verification process. This is a typical run-time scenario in situations where the user works with a particular host device for a period of time, and then moves-on with another host device.

In the migration dialogue, the list of available host devices is presented to the user via the migration dialogue menu, from which the user may make a choice. If migration is selected, the running application communicates with the remote migration service available at the target machine, and sends its *binary co*de and *current state* appropriately. Following, the remote migration service restarts the application, and requests its local application manager to inform all other application managers, at the various host devices, for the new device address of the running application that has just migrated.

4.2 I/O resource re-configuration procedure

During interaction, each application instance maintains the device address of the I/O resource services it uses, while it is also connected with the I/O resource registry for dynamically receiving notifications regarding detection or loss of I/O resources. Each application has a number of *active I/O tasks* (i.e., tasks initiated by the user in the current dialogue state), one of which, uniquely defined each time, is called the *focus I/O task*. The following situations may occur during run-time:

- When loss of I/O resources is detected, all the active tasks utilizing this resource should be re-configured (this includes the focus I/O tasks as well). It is possible that, for some of the active I/O tasks, no appropriate dialogue instantiation can be identified. This will practically mean that the particular application does not have adequate I/O to support interaction for those tasks. If this occurs for the focus I/O task, the undesirable state of a temporary *stall of dialogue* is reached, until the necessary resources become available.

- When discovery of I/O resources is detected, the state of the *focus application* is checked first. If its interface is stalled, its focus I/O task is directly re-configured, otherwise the current I/O configuration is maintained. As a second step, I/O re-configuration is applied to the rest of active I/O tasks in the focus application, and to all active I/O tasks (including the focus) in all running applications.

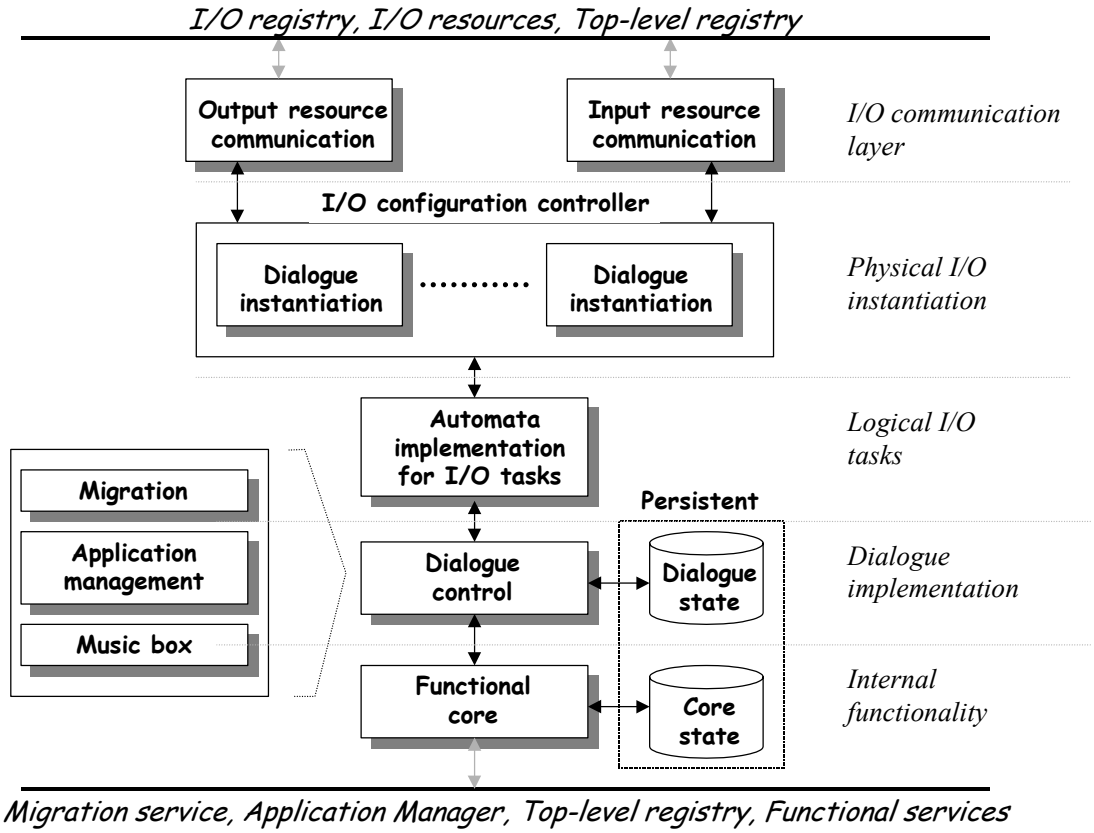

Figure 15: Distributed application *micro* architecture for interface and functional core implementation (internal composition of the "application instance" component of Figure 14).

4.3 Microscopic Interface architecture

The User Interface of a distributed application is, at run-time, a single system process (i.e., it is not a distributed system itself), controlling distributed I/O resources. Its implementation is based on a special-purpose architecture, emphasizing logical I/O control, dialogue state representation, redirection of physical I/O to external components, and dynamic I/O re-configuration. The role of each architectural module (see Figure 15 for an outline of the architecture) is discussed below:

- *Input / output resource communication*. Handles communication with the I/O resource registries (i.e., querying resources, gaining or releasing access), the utilized I/O resources (i.e., sending control commands, receiving notifications), and the top-level registry (i.e., requesting I/O resource registries).

- *Dialogue instantiation and I/O configuration controller*. This layer encompasses the implementation of the alternative dialogue instantiations of I/O tasks through the corresponding physical I/O resources (as defined in the dialogue design phase). Additionally, it selects for each active I/O task the most appropriate physical instantiation, according to the available I/O resources.

- *I/O task automata implementation*. This layer provides the implementation of dialogue automata, implementing logical dialogue control, for the various categories of I/O tasks, as they have been designed in the dialogue design phase.

- *Dialogue control*. The dialogue control implements the decomposition of the dialogue design for the three categories of functionality that a distributed application provides (as it is shown in Figure 4), based on the menu hierarchies presented within Figures 10, 11, 12 and 13. Part of the dialogue control is the dialogue state, which is kept up-to-date during run-time in a persistent form. The dialogue implementation supports start-up on the basis of a saved dialogue state file. This is necessary for the re-activation phase of migration, in which case the dialogue has to resume as if it was never interrupted, to ensure dialogue continuity.

- *Functional core.* This layer encompasses all the functionality that is not related to interface issues, and provides a simple API to the dialogue implementation, in such a way that the lower-level internal implementation details are hidden (e.g., MP3 player control, network download, storage, retrieving of application categories from the application manager, and migration requests). This is a typical separation policy met in UIMS systems (The UIMS Developers Workshop, 1992; Myers, 1995). Similarly to dialogue control, the state of the functional core is also kept in a persistent form, while its implementation supports re-activation based on a saved state file.

5. The User Interface Framework - *voyager* toolkit

The UI framework aims to provide a software library that provides to interface programmers all the necessary components for building interactive applications that utilise dynamically different types of I/O resources, as they are "on-the-fly", during interaction, discovered or lost. While pursuing for the right programming paradigm, there have been some key principles and objectives which have driven both the overall design and implementation of the programming paradigm, as well the more lower-level implementation choices:

Provide to programmers a higher-level toolkit. This practically means that we need to design and implement a "style" or "look & feel" analogy, in the form of a concrete toolkit library (Myers, 1995), as opposed to providing to programmers merely a library of low-level I/O resource APIs, with which they have to craft from scratch every new interface behaviour. As a typical development analogy, one might consider the case of GUI toolkits, where, instead of requiring that programmers are responsible for low-level display management, layout calculation, and input control, they have on their disposal a comprehensive GUI-toolkit library with higher-level reusable I/O behaviours (e.g. interaction objects like windows, push buttons, list boxes, etc.).

Hide and automate I/O resource configuration. It is crucial that programmers should not deal with I/O resource configuration, but this should be transparent, being automatically managed, and appropriately organised into various levels of abstraction. In particular, the client UI implementation code should not be dependent on the fact that I/O resource configuration takes place, unless the programmer requires to query and use specific types of I/O resources.

Provide a generic interface style. While acknowledging the need for a type of "look & feel", it is crucial to select one that can survive against the large diversity of I/O resources and inherent dialogue scenarios in the context of dynamic distributed I/O device resources. The "selector hierarchy and text entry" style has been chosen for a number of reasons. Firstly, it has been proved from past work (Savidis et al., 1996a; Savidis et al., 1996b) that it can be easily turned to genuine abstract object implementations, actually separating rendering and input control, from content management and callback notifications. Secondly, the selector object is powerful enough to easily model a variety of other dialogue behaviours, such as command buttons, radio buttons, check boxes, and confirm boxes. And finally, because it provides a compact API for UI implementation, it turns the construction of interfaces for our target domain an easily manageable task.

One abstraction to "rule them all" is a utopia. The selector paradigm works well for abstracting dialogues that, at the low-end, are implemented through distributed I/O resource primitives. However, when aiming to abstract GUI dialogues as well, it is clear that this paradigm is "less than perfect". The reason is that, when a GUI platform resource is available, one might expect an optimal use of the graphical interaction methods by supplying a full-fledged GUI interface version, rather than a mere simulation of device resources on the screen. But to accomplish such optimal use, the abstraction toolkit needs to reflect key GUI abstract properties, such as hierarchical structure and containment, while encompassing more comprehensive abstract dialogue behaviours (Desoi et al., 1989) like: command buttons, state buttons, and generic containers. Since the physical dialogue implementation resides on the device resource side (i.e. the GUI resource implements completely the I/O policy), interfaces implemented through such an interface toolkit will only manage to get physically instantiated

if and only if a "GUI resource" is available at run-time (i.e. no polymorphism, interfaces are bound to single specific resource class).

Moreover, unless lower-level graphical attributes are introduced on the abstract toolkit API, the interface programmer is unable to manipulate the graphical appearance of the resulting interface (e.g. fonts, colours, layout). However, if we decide to introduce graphical attributes in the API, we do make the interface toolkit directly dependent on the GUI paradigm, thus limiting the scope of such an abstract API to the GUI paradigm only. In conclusion, this argumentation reveals that *on the one side, the selector and text entry paradigm is indeed inappropriate for GUI resources, but on the other side the GUI toolkit abstraction is also inappropriate for distributed I/O device resources*. Previous work on toolkit abstraction through special-purpose development languages (Savidis & Stephanidis, 2001) has proved that to enable optimal interface control when dealing with style diversity, we need to support fusion of multiple styles in a policy defined as *multi-toolkit platforms* (Savidis et al., 1997; Savidis & Stephanidis, 2001b). Following those results, if GUI resources need to be utilised too, we need to provide a parallel GUI toolkit abstraction, fully inter-operable in programming with the selector style that depends on the presence of a GUI resource. This approach is the one we have employed and currently implement in the UI framework, and is discussed separately since it forms by itself an independent programming library with the rest of the framework.

Provide a device resource functional model for realistic use. Device resources, being remotely located services offering physical input or output interaction primitives, need to be designed in a way reflecting realistic use. In this context, our device resource functional model is not a prototype version for the purposes of the simulation, but reflects practical demands in the context of remote utilisation of interaction primitives. For instance, network connections may be broken, or the device resource itself may be stalled. Additionally, devices are expected to be made available by vendors with well-defined characteristics (i.e. properties), and to also define concrete communication protocol semantics. Resources may also provide extended functional APIs, that need to be formalised by means of a meta-API so that programmers can use all different APIs with the same programming conventions (like the property management approach the UI framework offers), while independent vendors may export resources by providing a concrete list of their properties, to enable be querying and matching during run-time, when seeking for the most appropriate devices.

Support a complete interface engineering model. Style toolkits for interface engineering should supply some common programming ingredients in order to facilitate the full power development of interactive applications. Some of the most typical features offered by the UI framework are: (a) input events and associated callbacks; (b) callbacks for logical actions (e.g. selected, edited, dialogue state changed); (c) dynamic focus control; (d) output handling through the implementation of RPC calls, supporting asynchronous notification; (e) abstract interaction objects, with automatically selectable input and out instantiations, adapting to the dynamically available physical I/O resources; (f) application main-loop supporting callbacks for injecting processing which is external to the UI framework.

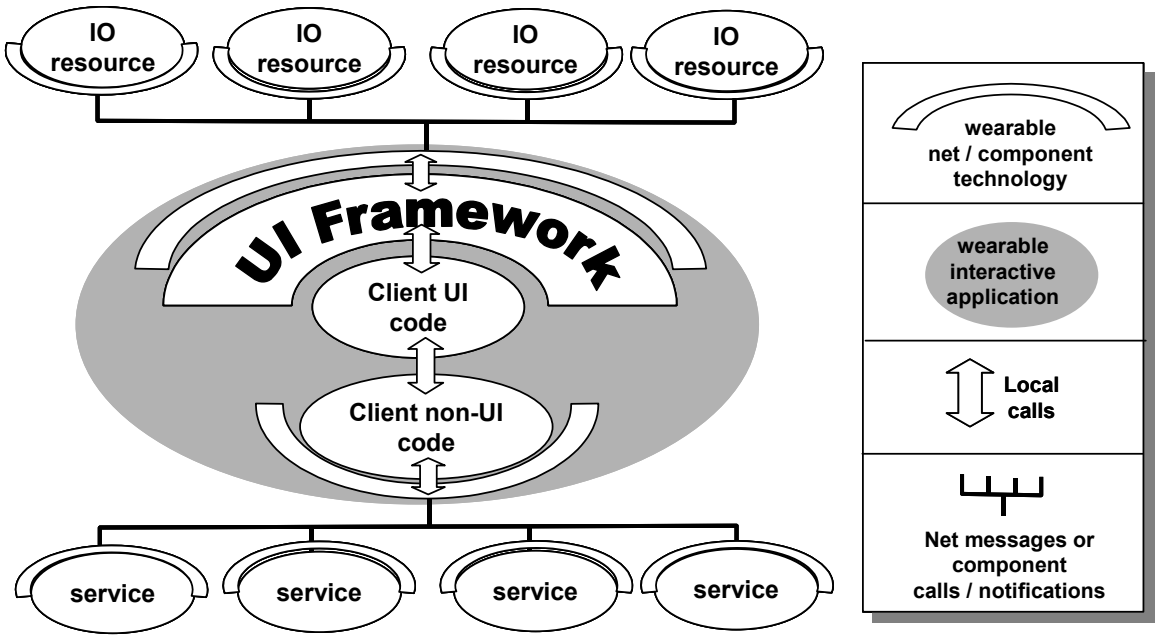

Figure 16: The role of the UI framework for building distributed wearable interactive applications supporting dynamic utilisation of remote I/O resources.

The UI framework (see Figure 16) has been used as an "umbrella" term to include: (a) the design paradigm, reflecting the selector and text entry style; (b) the specific implementation toolkit offered to programmers for building interfaces through abstract objects; (c) the device resource functional and description models, and all run-time algorithms to cater for dynamic I/O resource engagement; (d) the abstraction levels of the run-time architecture that neatly separates the abstract interface implementation from its dynamic physical form, the latter being independently configured during interaction as resources are engaged / disengaged; (e) the application manager connection for focus application control; and (f) the independent GUI proxy toolkit for dynamic GUI style utilisation. In figure 1-1, the role of the UI framework software library in developing distributed wearable applications is indicated. As it is shown, a distributed wearable User Interface is implemented as a typical client User Interface, as it is normally the case with the interface toolkits like MFC, or Motif, while the I/O resource distribution is handled underneath, through the distributed wearable communication (or component-ware) technology. In this report, the UI framework design is discussed, mainly focusing on the interaction paradigm and design method, while addressing some top-level software engineering issues (the detailed software design and implementation is the second phase of this research effort). Firstly, we start with a definition of the technical terms that are used throughout this report. Following, the intended software architecture is outlined, organised according to the various packages that form the implementation. Next, various key functional features of the framework are documented, together with the definition of some key algorithms (e.g. resource configuration, dialogue stall, IO task instantiation).

5.1 Definition of terms

Before proceeding with the detailed technical description of the UI framework and the various programming APIs it offers, the terminology that is used in this report is firstly defined.

Term	Definition
Device resource, or simply *Resource*	A remote exclusive-use service (i.e. not shared by many) providing input or output interaction facilities through a programming API.
Input Resource	A resource for input. If we do not qualify a resource explicitly as input, it means either an input or output resource.
Output Resource	A resource for output. If we do not qualify a resource explicitly as output, it means either an input or output resource.
Resource Description	A list of properties and values describing specific invariant features of the resource (e.g. size, location, containment in other device).
Resource Properties	A list of variant properties that can be altered dynamically through a programming API (e.g. setting LEDs, displaying titles).
Resource Group	A list of resources which have been made available together as belonging to the same named group (e.g. “local”, “room”, “clock”) instance. If resources are made available with an already existing group, this should not be considered as conflict, but be treated as if a new group instance is to be created.
Resource Granting	The process of requesting a resource for exclusive use, and accepting back and acknowledgement that the resource has been granted or not.
Resource Release	The process of requesting a resource to consider itself released by the requestor.
Resource Configuration	The process of seeking for the proper resources needed for the UI, possibly releasing resources not needed, and granting resources needed.
Network Link	An abstraction of a network communication API for asynchronous communication between two peers.
Protocol	The semantics that define the structure and type of content that is communicated between peers through network links.
Resource Responsiveness	The ability of a resource to respond to presence-test messages by acknowledging with the same message that it is still granted and working.
I/O Task	An abstraction of an activity the user can carry out, translated into an object-oriented programming API offering facilities for action notification. Links to the notion of interaction tasks (Foley et al, 1984) and virtual objects (Savidis and Stephanidis, 1995).
Selector Task	An IO task for the activity of selecting an option among a list of alternatives. Provides an API for option control and select notification.
Text Entry Task	An IO task for the activity of editing textual content. Provides an API for text-editing control and edit notification.
Task Input Instantiation	The binding of an abstract IO task to a specific input design and implementation class, supporting use input actions through the use of concrete input resources.
Task Output Instantiation	The binding of an abstract IO task to a specific output design and implementation class, supporting output delivery of the content through the use of concrete output resources.
Input style, Output style	An input instantiation or an output instantiation respectively, to indicate the direct physical interaction implementation input and output instantiations provide.
Instantiation Polymorphism	The ability of an IO task to have numerous alternative implementations for input and output instantiations.
Instantiation Ranking	The ordering of input and output instantiations within the implementation by increased preference.

Term	Definition
Focus I/O Task	The IO task instance with which the user performs interaction; this IO task is available to the user with the currently best possible input and output instantiation.
I/O Resource Manager	A service providing the registry for querying available resources.
Resource Discovery	The case where the IO resource manager notifies the UI framework that new device resources come to play.
Resource Loss	The case where the IO resource manager notifies the UI framework that some resources previously available have been lost.
Dialogue Stall	The situation in which for the focus IO task it is not possible to find the necessary input and output resources for any of its input or output instantiations.
Dialogue Revival	The situation in which, due to discovery of resources, we can find an input and output instantiation for focus IO task, thus exiting the dialogue stall state.
Application Manager	An application that is always running at each portable computing machine, providing a registry of all application classes that can be initiated, as well as a list of all application classes currently running. Enables the user to run / close applications, and also to give focus to applications as needed. ** This definition reflects the tentative design of the role of the application manager and may be updated as needed, as we move towards the detailed software design and implementation*.
Dialogue Suspension	The situation where the user, while working with an application, has selected to switch to the application manager. The application manager gains the focus, while the application dialogue enters a suspension mode (in effect, all resources for focus IO task are released).
Dialogue Resumption	The situation where the user, while working with the application manager, decides to give the focus to a particular application. The application manager enters a suspension state, the target applications enters a resumption state, and the IO task is configured to gain the resources needed.

5.2 Toolkit library organization

The UI framework is structured in a number of packages, with well-defined call dependencies among them (see figure 17). Some of those packages are to be used by: (a) the *interface programmer* to create applications, while (b) there are more advanced extensibility APIs enabling the *toolkit extension programmer* to extend either the range of I/O resources, or the collection of input and output instantiations for IO tasks, and (c) the *protocol programmer*, being the responsible to provide specific implementations of the protocol classes, corresponding to the introduction of extended classes for physical I/O device resources (by the toolkit extension programmer). The relationships among those packages, as well as the three fundamental programming roles are indicated in figure 17 and figure 18 respectively. Each of those packages corresponds to a specific sub-API, all of which are currently under construction.

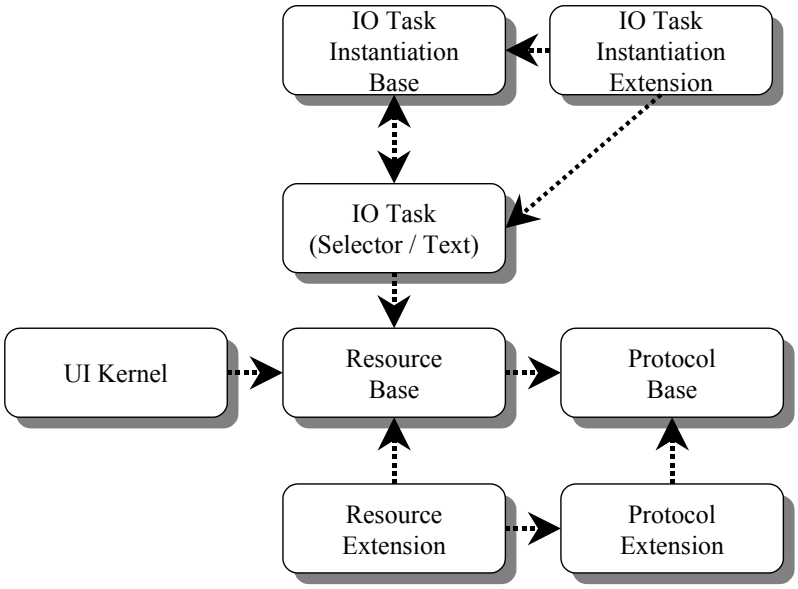

Figure 17: Package structure of the UI framework with top-level call dependencies.

Role	Uses packages	Extends packages
Interface programmer	IO Task Resource base Resource extension UI kernel	The interface programmer does not need to extend any package.
Toolkit extension programmer	ALL PACKAGES	Resource extension IO Task instantiation extension
Protocol programmer	Protocol base Protocol extension	Protocol extension

Figure 18: Programming roles relating to packages that need to be used, as well as to packages that can be extended.

5.3 Functional features and algorithms

In this section we briefly explain the functional properties of the UI framework to allow dynamic I/O resource utilisation and focus object re-configuration, enabling that the client programmer, when employing the abstract toolkit, is completely relieved from I/O resource allocation and management.

5.3.1 Focus IO Task configuration and effect in IO Task state

Focus IO task configuration is the process in which the UI framework has to release the current input and output instantiations of the focus IO task (for causes to be explained in the following sections), and subsequently identify and activate the new best possible input and output instantiations. The effect of such a configuration is that new resources come to play for implementing the input and output bindings of the focus object. *However, the state of the focus IO task and of all the various IO task instances existing at the point of configuration is not affected*. The reason is that each IO task instance encapsulates its own state, which is

completely independent of the input and output instantiations associated to the IO task (the latter used only for input control and output mapping). Going a little deeper to the software implementation, the IO task super-class uses dynamically associated instances of input and output instantiation classes, enabling on-the-fly update with by assigning different instances of input and output instantiation styles, without an inherent effect on the IO task instance itself. This reflects an adapted version of the "state" OOP pattern, particularly suited to abstract interface objects, known as the "dynamic polymorphic I/O" pattern, having its origins in the implementation of the I-GET UIMS (Savidis and Stephanidis, 2001).

5.3.2 Resource discovery

If dialogue is suspended, no action is taken, since the application does not have the focus. When new resources are discovered, the UI framework seeks for the best possible input and output instantiations, assuming the resources available are: (a) the ones used currently by the focus (i.e. they are recycled); (b) the resources just discovered; and (c); all other, not granted, available resources. If the best instantiations found are the ones currently in use, no actions are taken. Else, the current input and output instantiations are released (only if dialogue was not stalled, else those instantiations are null), while the currently best found input and output instantiations are activated (*when output instantiations are activated, or when they are associated with a different IO task, they call their display method automatically*). In case the dialogue was previously in a stalled state, and now is in a working state, dialogue revival is signalled.

5.3.3 Resource loss

If dialogue is suspended, no action is taken, since the application does not have the focus. When resources are lost (either due to network failure, or due to lack of responsiveness, or because the services are gracefully removed), if the dialogue is already stalled, no further action is taken. Else, the UI framework asks the current focus object if it happens to use any of the lost resources. If not, no action is taken. Else, the current input and output instantiations are released, and a configuration round is entered, seeking for the best possible input and output instantiations. If no such instantiations can be found with the currently available resources, the dialogue is signalled as stalled, else, those found input and output instantiations are activated.

5.3.4 Focus object change

If dialogue is suspended, no action is taken, since the application does not have the focus. If the focus object is changed, and the previous focus object is of the same IO task class with the new focus object, if the dialogue is not stalled, the input and output instantiations of the old focus object are just associated with the new focus object, and no further action is taken. If the new focus is of a different class with the old focus, the input and output instantiations are released (only if the dialogue is not stalled), and a configuration procedure (as before) for the focus object is performed. If this results in no viable input and output instantiations, and the dialogue was previously working, we signal dialogue stall. If we do find viable instantiations and the dialogue was previously stalled, we signal dialogue revival.

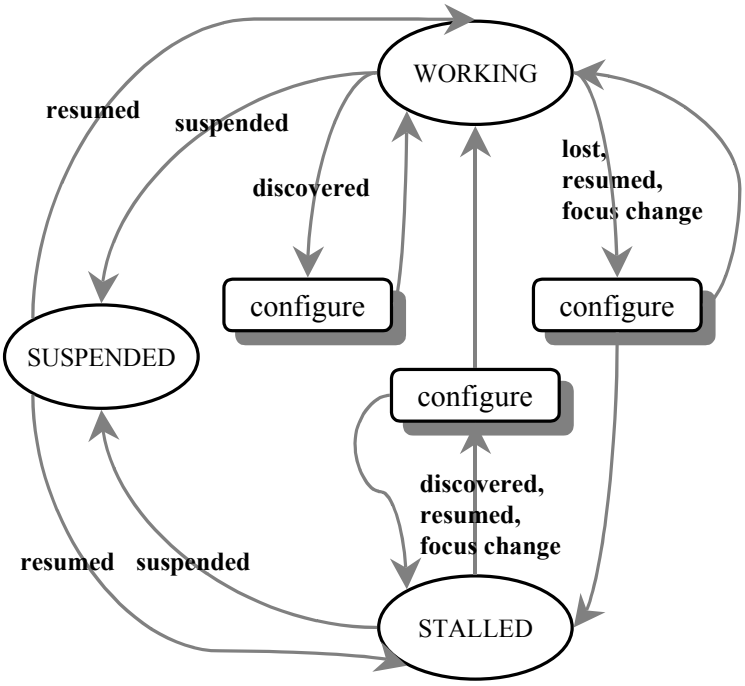

Figure 19: Dialogue state change automaton.

5.3.5 Dialogue state change

The state change logic of the UI framework is shown with a state automaton in figure 19. In all transitions resulting in state changes, the corresponding callbacks are called (i.e. suspension, resumption, stall and revival callbacks).

5.3.6 Resource grouping

Normally, resources are made available "as they are", without any particular restrictions on their employment, apart from those related to matching criteria that concern resource descriptions. Hence, to retrieve a resource registered with the IO resource manager, all that is needed is to find an available resource matching the requested type, specific class id, and list of property / value pairs. However, there are scenarios where:

- The designer of input or output instantiations would like to pose some restrictions regarding the choice of resources as a whole (not in isolation), such as requiring all resources to located on the same computing machine (e.g. be local);

- The vendors of collections of I/O resources may wish to relate particular resources together, so that programmers can distinguish if resources belong to the same group instance. For instance, imagine the case where vendors export complete IO task implementations, on specific devices, e.g. a "menu by X vendor", by means of a pair of one "menu input" and one "menu output" resource. In case interface programmers wish to extend the instantiations of the selector task to cater for this case too, all that is needed is to define one more input instantiation class requiring the "menu input" resource, and another output instantiation class requiring the "menu output" class. This works fine, except one important remark. If, during run-time, we have multiple such "menu" resources, there is no way for the UI framework to ensure that it takes both from the same device. Here comes the grouping ability to resolve this issue. To

ensure the required pair of instantiations is taken from the same device the following are required:

- The vendor has to register the pair of such “menu” resources for each different “menu” device together to the IO resource manager, qualifying those with the same group id, say “menu”;
- Within the instantiations, the programmer explicitly sets within the “menu” input and output resource descriptions the group id to be the one used to export the resources, i.e. “menu”.

When resources are registered in groups, the IO resource manager creates a new group instance; it is legal to have group instances with the same group name (see figure 20). If no group is defined, the resources are added within the same single group instance, with name “default”. The above scenarios are covered successfully by the UI framework due to the fact that the resource allocator satisfies the rule defined below, when groups are defined within the resource descriptions of input and output instantiations:

An input instantiation I and an output instantiation O with required resources IR and OR are viable if and only if all resources from IR ∪ OR that have the same group id, are found at run-time within the same group instance.

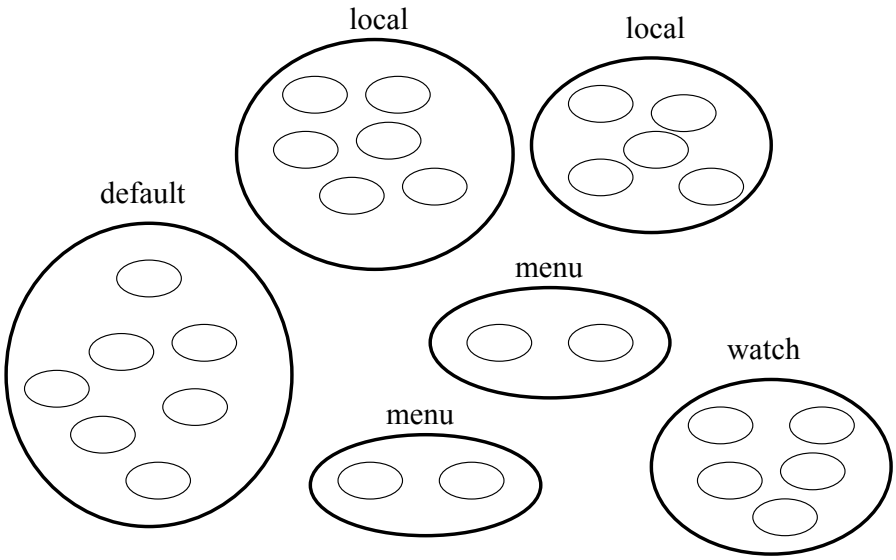

Figure 20: Supporting group instances in the IO resource manager of the UI framework.

5.3.7 Hardware and software resource properties

The distinction between hardware (invariant) and software (variant) device resource properties is fundamental. The first concerns capabilities the device offers which can be declared as criteria for resource request and allocation within input and output instantiations (i.e. we ask for resources necessarily having the defined properties). The second concerns a type of extended control API the device resource offers for controlling its use and manipulating its parameters. Some software properties will not be supported by some devices. For instance, not all button device resources will offer a “title” property. Hence, it is up to the toolkit extension programmer to query the available properties, based on the

documentation of the device resources as provided by the vendors, and use those properties maximally for the benefit of interaction quality.

At this point one might argue that there is no need for hardware properties at all; the programmer could simply define resource type and class id only, while query and use all properties dynamically inside the instantiations. In this case, after discovery, an additional round of negotiation with all candidate resources would be required, in order to seek for those with the best available properties. This make the life of the toolkit extension programmer far more difficult, while it may require temporary granting (unless we allow property management requests to device resources, even when they are not granted by the caller), and direct communication for querying available properties, until the final most appropriate resource collection is identified. Clearly, this adds too much unnecessary implementation complexity to the allocation round, on behalf of the toolkit extension programmer role. Instead, based on invariant properties, resources simply declare explicitly upon registration what they can offer, while different instantiations define accurately with resource descriptions what they need, enabling the UI framework to identify the best possible instantiations based on property querying and matching.

Hardware properties	
Name	**Value**
“shape”	“oval”, “circle”, “rectangle”
“size”	“small”, “medium”, “large”
“hastitle”	“yes”, “no”
Software properties	
Name	**Value**
“title”	string
“fg”	string, “<RGB>, <RGB>, <RGB>”
“bg”	string, “<RGB>, <RGB>, <RGB>”
“font”	string, “<family>, <points>”
Input Instantiation Style 1, Selector Task, Ranking 1 (top)	
NEXT button	**SELECT button**
• Type : ResourceTypeInput • Class : ButtonInputResourceClass • Group : “localdevice”	
“shape” “indifferent”, “hastitle” “yes”, “size” “medium”	“shape” “indifferent”, “hastitle” “yes”, “size” “large”
Input Instantiation Style 2, Selector Task, Ranking 2 (bottom)	
NEXT button	**SELECT button**
ResourceTypeInput, ButtonInputResourceClass, “localdevice” group.	
“shape” “indifferent”, “hastitle” “indifferent”, “size” “indifferent”	“shape” “indifferent”, “hastitle” “indifferent”, “size” “indifferent”

Figure 21: Example with hardware and software properties, and role in descriptions for resource matching within input and output instantiation styles. One of the planned extensions

regarding hardware property matching, is to allow multiple matching values (e.g. can have "size", with either "small" or "medium" being matching values).

One example is provided in figure 21, showing the distinction among software and hardware properties for a button input resource. The hardware properties can be engaged in different input instantiations with different values. For instance, for the selector input instantiation extension, the toolkit extension programmer may define two alternative input instantiations, ranked appropriately, as shown in figure 21. This explicit hardware property description enables the UI framework to choose the best possible (i.e. with higher ranking) by matching property names and values. On the contrary, the software properties are used in the instantiation implementation dynamically (if they exist, they are used, else, not).

5.3.8 Application focus change and failure scenarios

Switching to the application manager can be initiated either by an application (normal switching) or by the application manager (enforced switching) - see figure 22. Similarly, an application may ask to be resumed explicitly, in which case the application manager may give the focus back (send a resume request).

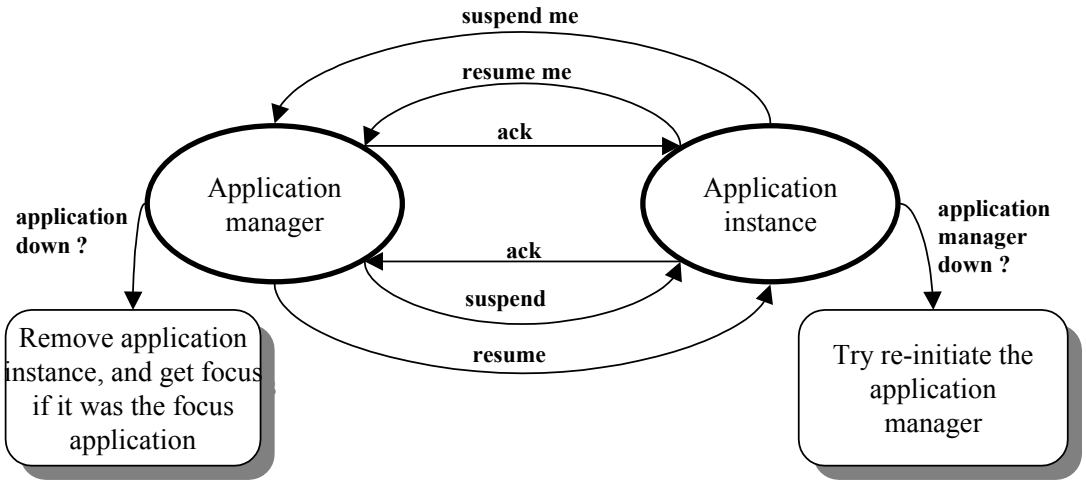

Figure 22: Application focus switching, having the application manager in charge.

In case the application manager fails (e.g. a system crash), the running applications will try to initiate it. To avoid multiple initiations, this will be implemented through an initiation mutex. If there is no application running, and the application manager goes down, the way the user will able to re-run the application manager is operating system dependent.

5.3.9 Device resource communication

Here, we will schematically outline the communication requirements with device resources (i.e., all types of resources - see figure 23). Resources reply to the sender either immediately or after the processing requested has finished. Only the release request does not need acknowledgement. In all other requests, the UI framework performs busy waiting of the response with time outs. In case the time out exceeds, the call returns with a returned value indicating failure, while the resource itself is considered as failing and the IO resource manager is subsequently informed.

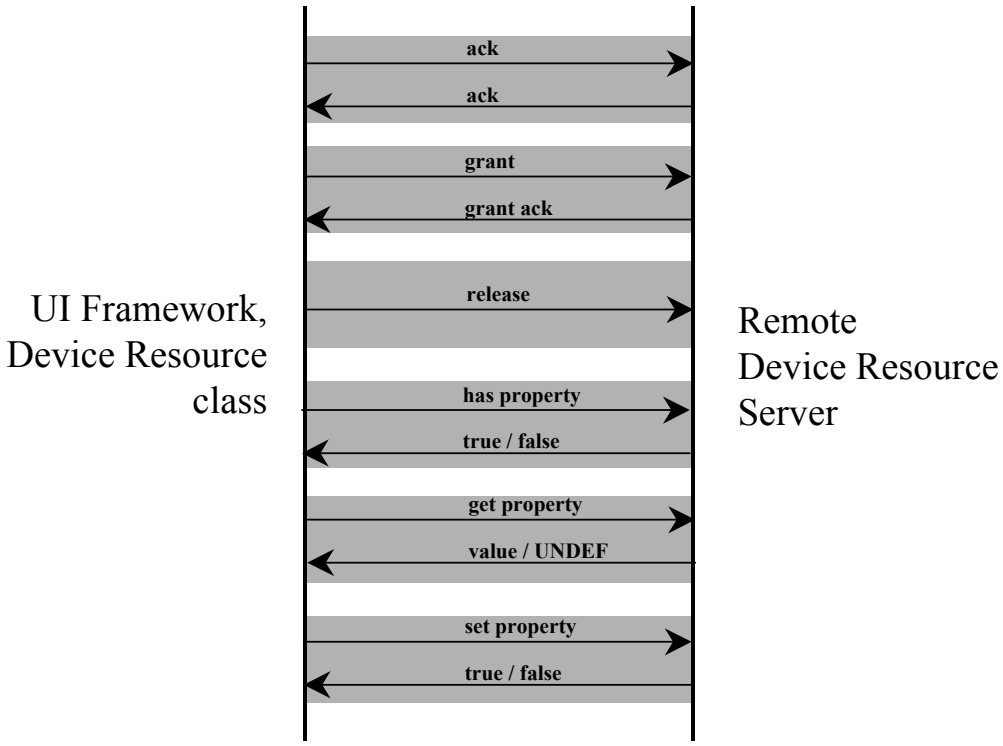

Figure 23: Messages and corresponding responses in communicating with device resources.

5.4 The GUI abstraction Toolkit

5.4.1 Proxy GUI Object Classes

We have identified the following generic GUI classes that will be offered by the GUI proxy toolkit, and mapped remotely to concrete GUI classes:

- Container class, which will cover cases such as: frame (top level, modal / non-modal), scroll view, panel), with explicit category qualification.
- Push button
- Radio button (enabling radio groups)
- Check box
- Text field (single text line)
- Text area
- List box

The layout policy will reflect a grid-based (tabular) layout manager, supporting layout constraints. The overall design of the various classes, relevant attributes and functions to be exported by the toolkit is still tentative and should by no means considered as finalised.

5.4.2 Generic Toolkit Interfacing Protocol (GTIP)

GTIP is an application-level protocol introduced in (Savidis et al., 1997), for remotely connecting to toolkits, thus enabling the remote separation of the actual toolkit library with the client program, in an analogous way the X lib separates X clients from the X server. The

GTIP enables the same separation to be implemented for Xt as well. GTIP has been implemented in the context of I-GET UIMS (Savidis & Stephanidis, 2001), and served as the backbone for integrating and connecting to different toolkits like Xt/Xaw, Windows MFC and HAWK. The GTIP is generally based on the following communication features:

- From remote client → toolkit server
  - Object instantiation / destruction requests
  - Property modification requests
  - Output events
- From the toolkit server → remote client
  - Property modification notifications
  - Input event notifications
  - Returned parameters (from output events)
  - Object method notifications

Due to the fact that we cannot perform a direct porting of GTIP from the I-GET UIMS, we will implement it in the context of the UI framework. It should be noted that the separation of the client from the toolkit library enables the "thin interface client" concept, something which is particularly critical for wearable applications, since, a thin interface client may run on a portable machine, but display a fully interactive GUI on a dynamically available desk-top machine (the latter requires availability of the whole native GUI library and respective components).

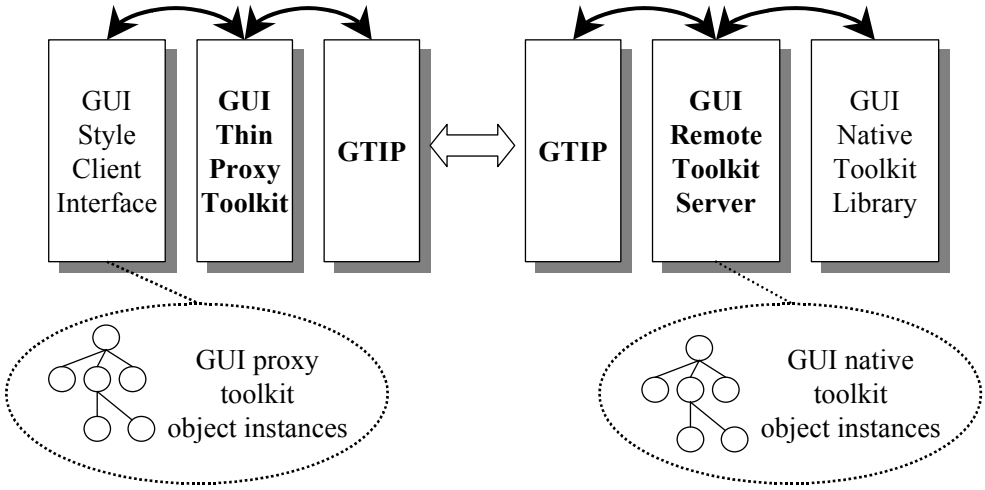

Figure 24: Run-time architecture shown the links among the GUI proxy toolkit, GTIP and GUI toolkit server.

5.4.3 GUI Toolkit Server Resource

The GUI toolkit server will be implemented as a native GUI client, dispatching all requests originated from the remote client, while posting notifications back as needed (such as user input events, method notifications such as button presses and text editing) – see 24 for the run-time architecture.

5.5 Future Extensions

There are specific extensions to the UI framework planned, aiming to maximize the flexibility and effectiveness of the framework in managing device resources, while also enabling toolkit extension programmers to choose among alternative policies that the UI framework offers for identification and activation of best possible input and output instantiation styles. More specifically, the most representative extensions under investigation concern:

- Associating input and output styles together in a form of "requires style" relationship. In practical terms, we want to enable programmers to define cases where: *a specific input style I is activated if and only if a specific output style O is also activa*ted. This enables input and output styles to be associated together, by forcing that there can be no case in which one of those is activated, on the absence of the other (i.e. *both of them, or none of them*).
- Enabling device resource vendors to insert on-the-fly resources within specific resource group instances. According to the present implemented policy, resources are exported either in isolation or in groups, meaning in the latter scenario, a new group instance is always created. Hence, currently, there is no way for "late insertion" of an additional resource in an already existing group instance. Such an extra feature is necessary in the following scenario: a group of resources is made available to the IO resource manager; after a period of time, some of those resources fail (e.g. network broken, or the services are stalled); the *resource exporting module* (i.e. a type of local service directory for the whole exported group) re-initiates the failing resources and inserts those within the same group instance. This approach requires:
    - that there are explicit resource exporter components, being responsible for communicating with the IO resource manager and informing for the availability of device resources (i.e. discovery / loss means an exporter declares availability / removal of resources) ; and
    - that each resource exporter receives an external run-time group reference by the IO resource manager, for each exported group of resources; then, for late insertion, the exporter supplies to the IO resource manager the list of new resources together with the appropriate group reference.
- Extending the logic for resource property matching by enabling values to fall within a list of legal values, or to be matched through a programmer defined filter function. This extends the present policy, which enables either *single* value matching, or *any* value matching (if the reserved value *"indifferent"* is supplied). Through this extension, we will enable the implementation of more sophisticated style matching design logic.

6. Conclusions

The development of everyday electronic appliances is facing a paradigm shift. Hardware vendors are increasingly adopting the approach of supporting embedded operating systems and building network-ready devices, practically turning the development of interactive facilities and services on such devices into a software development process. This trend, while initially envisioned in the context of smart-home technologies, has been mainly pushed forward within the last few years due to the large competition and fast progress in the domain of mobile phones. Effectively, concepts such as ubiquitous computing, which have been introduced in the beginning of the '90s (Weiser, 1991), become now concrete technical targets, since both the required software development technologies (embedded operating systems, distribution and inter-operability technologies, the Web), as well as the hardware infrastructure (wireless networks, more powerful and network-ready computing devices, location-awareness technologies) are currently available. In this context, there are some basic properties, which heavily affect the way interactive software has to be developed in this new infrastructure:

- *User mobility*, therefore, *mobile interface*. The user is moving within an open computational environment, far beyond the desktop set-up, requiring continuous availability of (migrating) software applications "on the move".

- *Distributed I/O resources*, therefore, *distributed interface*. Different computational resources, such as interactive or functional devices, are physically positioned at different locations. Effectively, by splitting the interface into various distributed collaborating components, "the environment becomes the interface".

- *Dynamic I/O resource presence*, therefore, *dynamic interface*. Due to user mobility and device mobility (hand-held devices or environment robots), the availability of I/O resources to mobile applications varies with location and time. Consequently, the interface is required to dynamically utilize the I/O resources available in each particular case. Therefore, the interface changes "on the fly", while preserving dialogue continuity.

In this context, a special-purpose application experiment has been carried out, aiming to emphasize the three key properties mentioned above, pursuing an appropriate interface design and software engineering approach that would particularly fit into this problem domain. It has been considered more appropriate to construct an application scenario that would engage everyday tasks, as opposed to traditional office tasks (such as text processing or file management), since this would better match the concept of ubiquitous computing. The "music box" application scenario has been designed, which constituted the test-bed for the overall development efforts. The results and contributions of this work mainly fall into three categories:

- *The interface design approach*. The technique employed for the design of dialogue artefacts displays a combination of: (a) interactions tasks (Foley et al, 1984); (b) abstract objects with polymorphic dialogue instantiation (Savidis and Stephanidis, 1995); and (c) the use of state automata (Jacob, 1988) for describing the dialogue of interaction objects. Although no new design method is practically introduced, the way

the key ingredients from those techniques have been engaged in this particular design domain has proved to be rather powerful.

- *The interface design artefacts*. Firstly, the outcomes of the design process revealed the need to organize the dialogue in three categories: application-specific dialogues, application management, and distribution control (migration and I/O re-configuration). Secondly, the evaluation process has shown that the use of menu-hierarchies with modal dialogues, and support of a single focus I/O tasks, is an appropriate design choice, since the engaged I/O resources were not appropriate for concurrent dialogues. Finally, the need to engage location parameters and distance constraints in the dialogue instantiation logic of I/O tasks has been identified, since, due to the particular physical installation of I/O resources, the dialogue continuity may be disrupted.

- *The software engineering method*. The key engineering challenges, namely to support multiple applications, distributed / dynamic / shared I/O, and application migration, were addressed in two architectural levels: the macro-level architecture, identifying common components to provide the system infrastructure for distributed applications (see Figure 14), and the micro-level architecture, providing the internal decomposition of an application instance (see Figure 15).

During the evaluation phase, strong emphasis has been put in enabling users to understand the particular dialogue context, assimilate the supplied feedback and be aware of the types of activities possible at any point in time (see Figure 5). These design issues directly link with the principle of continuity, which becomes a key objective in the domains of ubiquitous computing and nomadic interaction. Additionally, it is argued that more efforts need to be undertaken to systematically analyze various categories of user tasks in the context of ubiquitous computing, aiming to identify concrete continuity-oriented dialogue design requirements, linking directly with the physical design.

The initial application scenario has driven concrete efforts towards the software design and implementation of a toolkit to provide: (a) a concrete "look & feel" based on selector and text-entry object classes; (b) encompass I/O remote resource control through heavyweight classes hiding all networking details; and (c) abstraction layers defining implemented alternative ways of binding the abstract object classes to concrete I/O resource utilization. This process is well in progress and the first full-fledge version of the toolkit has been already produced and is subject to intensive testing with simulated remote resources.

Acknowledgements

Part of the work reported has been partially funded by the European Union under the following projects and we acknowledge the co-operation of the partners involved:

(i) "Theory and Applications of Continuous Interaction Techniques" (TACIT). Project's web site: http://kazan.cnuce.cnr.it/TACIT
(ii) "A Runtime for Adaptive and Extensible Wireless Wearables" (2WEAR - IST-2000-25286). The partners of the 2WEAR Consortium are: Foundation for Research and Technology – Hellas, Institute of Computer Science, Greece (Prime Contractor); Nokia Research Center, Finland; Swiss Federal Institute of Technology - Zurich Institute for Computer Systems, Switzerland; MA Systems and Control Limited, UK.

Part of the work reported has been supported by national government funding in the context of "Excellence in the Research Institutes Supervised by the General Secretariat for Research and Technology", under the project "Excellence ΚΠΣ 0006".

References

Abowd, G., Mynatt, E. (2000*). Charting Past, Present, and Future Research in Ubiquitous Computing*. ACM Transactions on Computer-Human Interaction, 7(1), March 2000, 29-58.

Bharat, K., Cardelli, L. (1995). Migratory Applications. In *Mobile Object Systems: Towards the Programmable Internet*, Vitek, J., Tschudin, C. (Eds), Springer-Verlag: Heidelberg, Germany, 131-148.

Calvary, G., Coutaz, J., Thevenin, D., Rey, G. (2001). Context and Continuity for Plastic User Interfaces. . In *Proceedings of the I3 Spring Days Workshop on Continuity in Future Computing Systems*, Porto, Portugal, April 23-24, 2001, CLRC, 51-69.

Cardelli, L. (1995). Obliq : a language with distributed scope. *Computing Systems 8(1)*, MIT Press, 27-29.

Desoi, J., Lively, W., and Sheppard, S. (1989). Graphical Specification of User Interfaces with Behavior Abstraction. In *Proceedings of the CHI'89 Conference on Human Factors in Computing Systems* (Austin, Tex, April 30-May 4, 1989), 139-144.

Duke, D., Faconti, G., Harrison, M., Paterno, F. (1994). Unifying view of interactors. *Amodeus Project Document*: SM/WP18, 1994.

ERCIM NEWS (2001). Special Theme: *Ambient Intelligence*, Issue 47, October 2001, ERCIM. [On-line]. Available at: http://www.ercim.org/publication/Ercim_News/enw47/.

Foley, J. D., Wallace, V. L., Chan, P. (1984). The human factors of computer graphics interaction techniques. IEEE Computer Gr. & Appl, 4, 11 (November 1984), 13-48.

Ishi, H., Ullmer, B. (1997). Tangible bits : Towards seamless interfaces between people, bits and atoms. In *Proceedings of the ACM SIGCHI 1997 Conference on Human Factors in Computing Systems* (March 22-27), Atlanta, GA, 234-241.

Jacob, R. (1988). An executable specification technique for describing Human-Computer interaction. *In Advances in Human-Computer Interaction, 1*. Hartson, R (Ed). Ablex Publishing, New Jersey, 1988, 211-242.

Massink, M., Faconti, G. (2001). A Reference Framework for Continuous Interaction. In *Proceedings of the I3 Spring Days Workshop on Continuity in Future Computing Systems*, Porto, Portugal, April 23-24, 2001, CLRC, 1-23.

May, J., Buehner, M., Duke, D. (2001). Continuity in Cognition. In *Proceedings of the I3 Spring Days Workshop on Continuity in Future Computing Systems*, Porto, Portugal (April, 23-24), 2001, CLRC, 39-50.

Myers, B. (1990). A new model for handling input. *ACM Trans. Inform. Syst. 8, 3* (July 1990), 289-320.

Myers, B. (1995). User Interfaces Software Tools. *In ACM Trans. on Human-Computer Interaction, 12(1)*, march 1995, 64-103.

Norman, D. (1990). The problem with automation: inappropriate feedback, not over-automation. In Broadbent, D., Reason, J., Baddeley, A. (Eds), *Human Factors in Hazardous Situations*, Clarendon Press, Oxford, UK, 137-145.

Norman, D. (1998). *The Invisible Computer*. MIT Press, Cambridge, MA.

Savidis, A., & Stephanidis, C. (2001a). The I-GET UIMS for Unified User Interface Implementation. In C. Stephanidis (Ed.), *User Interfaces for All - Concepts, Methods, and Tools* (pp. 489-523). Mahwah, NJ: Lawrence Erlbaum Associates (ISBN 0-8058-2967-9, 760 pages).

Savidis, A., & Stephanidis, C. (2001b). Development Requirements for Implementing Unified User Interfaces. In C. Stephanidis (Ed.), *User Interfaces for All - Concepts, Methods, and Tools* (pp. 441-468). Mahwah, NJ: Lawrence Erlbaum Associates (ISBN 0-8058-2967-9, 760 pages).

Salber, D., Dey, A., Abowd, G. (1999). The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proceedings of the ACM SIGCHI 1999 Conference on Human Factors in Computing Systems* (May 15-20), Pittsburgh, PA, 434-441.

Savidis, A., Stephanidis, C., Korte, A., Crispien, K., Fellbaum, K. A (1996a). Generic Direct-Manipulation 3D-Auditory Environment for Hierarchical Navigation in Non-visual Interaction. In proceedings of the *ACM ASSETS'96 conference*, Vancouver, Canada, April 11-12, 1996, 117-123.

Savidis, A., Stephanidis, C. (1996b). Unified manipulation of interaction objects: integration, augmentation, expansion and abstraction. In *proceedings of the 3rd ERCIM Workshop on User Interfaces for All*, November 3-4, INRIA Lorraine (1996), 75-89

Savidis, A., Stephanidis, C., & Akoumianakis, D. (1997). Unifying Toolkit Programming Layers: a Multi-Purpose Toolkit Integration Module. In M.D. Harrison, & J.C. Torres (Eds.), *Proceedings of the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS '97)*, Granada, Spain, 4-6 June (pp 177-192). Berlin: Springer-Verlag.

Savidis, A., Stephanidis, C. (1995). Developing Dual Interfaces for Integrating Blind and Sighted Users: the HOMER UIMS. In *proceedings of the ACM SIGCHI 1995 Conference in Human Factors in Computing Systems* (May 7-11), Denver, CO, 106-113.

The UIMS Developers Workshop. (1992). A meta-model for the run-time architecture of an Interactive System. *SIGCHI Bull 24, 1* (January 1992), 32-37.

Weiser, M. (1991). The computer for the 21st century. *Scientific American 265(3)*, 94-104.

Wise, G. B., Glinert, E. P. (1995). Metawidgets for multimodal applications. In proceedings of the *RESNA'95 conference*, Vancouver, Canada, June 9-14, 455-457.

Wellner, P. (1990). Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. In *Proceedings of the AC CHI'89 Conference on Human Factor5s in Computing Systems*, (April 1-5, 1990), Seattle, WA, 177-182.