VitKit: a Voice Interaction Toolkit

Mike Rizzo, Peter Linington and Ian Utting
Computing Laboratory, University of Kent at Canterbury,
Canterbury, KENT CT2 7NF, UK

September 12, 1994

1 Introduction

The modern tone-dialling telephone is often under-estimated as a vehicle for user interfacing with applications. While its input and output capabilities are a far cry from even the simplest of today's mouse-keyboard-display terminals, the extensive public telephone network gives the telephone a big advantage with respect to availability. Suitably constructed applications can be made accessible to anyone from virtually anywhere, be it at home, at work, on a train, or even abroad. From a usability point of view, experiments and real systems have demonstrated that carefully constructed telephone-based user interfaces can be successfully applied to a variety of areas [G$^+$87], [GB83], [Sch93], [Res93]. However, the scope of these applications has to a large extent been limited to localised message systems exhibiting little or no integration with other data-oriented applications.

Additionally, little has been published on the actual specification and implementation of telephone-based interfaces. To date most research appears to have concentrated on high-level specifications making use of visual programming [Rep92], table-oriented techniques [RBG86], and form-oriented techniques [Res92]. All these have proved to be suitable for building bulletin-board/messaging systems with static interfaces, but are rather inflexible as to how they interact with more complex applications.

This paper describes the Voice Interaction ToolKIT(VitKit), a C++ class library for building telephone-based user interfaces. Rather than use a high-level specification approach, it is intended that programmers use the classes directly to compose interfaces, although the possibility of developing code-generators for building (parts of) interfaces is not excluded. The toolkit supports dynamic construction and re-configuration of interfaces and adopts a very flexible approach to interfacing with underlying applications.

The rest of the paper is structured as follows: section 2 describes the motivation for VitKit, section 3 outlines the approach taken, section 4 describes the underlying architecture, section 5 discusses the class hierarchy and section 6 describes how VitKit was used to construct interfaces to a highly-configurable voice message system. Finally section 7 concludes with some observations and an outline of future directions.

2 Motivation

VitKit was developed as a tool for the Open Distributed Office (ODO) project [RLU94b] at the UKC Computing Laboratory. This project aimed to exploit integration of voice and data services in a distributed office environment.

For example, callers can make appointments via phone-based 'automated secretary' services, and users can browse their calendar of events via the phone. A user's calendar data might also be used as a source of location to automatically route incoming calls to the nearest access point to that user.

Another feature of ODO is that users are able to offer alternative fallback services when they are not able to answer calls. For example, if a site's postmaster was going to be away for a few days, he might specify that all calls related to electronic mail issues should be forwarded to the deputy postmaster, whereas all other calls should be re-directed to his voice message system. Thus callers would be greeted by a menu, asking them to select which of the two options they would like to take.

Accordingly, the tools for building telephone user interfaces had to (i) provide flexible mechanisms for interfacing with applications and, (ii) allow for dynamic construction of interfaces. For example, the automated secretary user interface interacts with the calendar application at two different points: it needs to retrieve time-slot availability information, and then needs to create an appointment. Both involve making a number of remote procedure calls. The second example ie alternative fallback services, requires the ability to dynamically build interfaces from some input specification.

These two characteristics might also be required together. For example, selection of an item from a voice menu may result in a call to a potentially remote service, the results of which could be used to determine the next menu, or even to add options to the existing one.

VitKit was designed with these desirable characteristics in mind. Although intended for use in the ODO project, it can easily be applied to other application areas.

3 Approach

The design of the toolkit mimics the concepts normally associated with graphical interface toolkits[1]. Thus, for example, there are the notions of an 'interaction technique' or 'widget' and a 'composition mechanism'. The term 'vidget' (voice widget) is used as the voice equivalent of the graphical widget.

A fundamental difference between audio and graphical interfaces lies in the dimensions through which information is conveyed and input is received. In graphical interfaces, input and output is done over a two-dimensional space. In voice interfaces there is only one dimension, namely time. VitKit provides mechanisms which takes this into account. For example, while graphical interfaces provide composition mechanisms to stack interaction techniques vertically or horizontally, VitKit provides mechanisms to execute interaction techniques sequentially or concurrently.

Compositions of vidgets are themselves vidgets. These composite vidgets provide methods to add and remove vidgets from within them; program code can use these methods to reconfigure a composition dynamically.

Interfacing to applications is done via action vidgets. These may be placed anywhere in a composition and invoke some specified application function when encountered in the course of execution.

Before proceeding with details of the interaction techniques themselves, an understanding of the supporting architecture is required. The next section outlines the framework within which

[1]Some aspects of VitKit's design have particularly been influenced by the InterViews toolkit [LVC89].

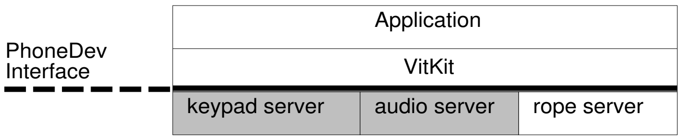

Figure 1: Architecture of VitKit applications

VitKit is intended to be used.

4 Architecture

VitKit uses components developed as part of the Palantir project at UKC. This project sought to provide a multimedia development infrastructure in a distributed environment. The emerging ISO reference model for Open Distributed Processing (RM-ODP) [Int][2] was used as a framework and components were implemented using ANSAware [Arc92]. The latter provides engineering support for building distributed systems. Amongst other things, it includes mechanisms for multiple concurrent threads and remote procedure call.

The Palantir audio system uses a standard set of interfaces to encapsulate a variety of audio-capable hardware types (notably the audio hardware available on HP and SUN workstations). Implementations of such interfaces are known as audio servers. Palantir also provides an audio storage server and a rope server. A rope is composed from segments of samples held by the storage server. The rope server interface provides operations to record and play ropes, both of which may return progress information to the client via supplied callbacks. Further details of the Palantir audio components can be found in [Li94].

In addition to the audio and rope servers, VitKit makes use of a keypad server to receive keypress events. Viewed from this angle, the objective of VitKit is to provide a framework which enables the user to control the rope server via a keypad.

VitKit applications run on top of a virtual machine known as the 'PhoneDev'. This establishes a uniform interface for which different implementations can be provided in order to enable different underlying technologies to be used. It also represents a potential distribution boundary. For example, one implementation of the PhoneDevice was constructed for workstations equipped with audio hardware. This makes use of a Palantir audio server (audio I/O) and an X window based keypad server. Another implementation was built for ordinary telephones on the POTS network. This makes use of a PC-based interface to the local POTS network. The interface digitises incoming voice, plays digitised voice out onto the POTS network, and detects keypad tones. It is capable of making outbound calls and handling incoming calls. All the interface's functionality is encapsulated within a single ANSAware object that fills the roles of both audio server and keypad server.

Figure 1 illustrates the organisation of components involved in a VitKit application. The shaded areas denote components that can be replaced by others to provide a different implementation of the PhoneDevice.

[2] For a more gentle introduction to RM-ODP read [Lin92].

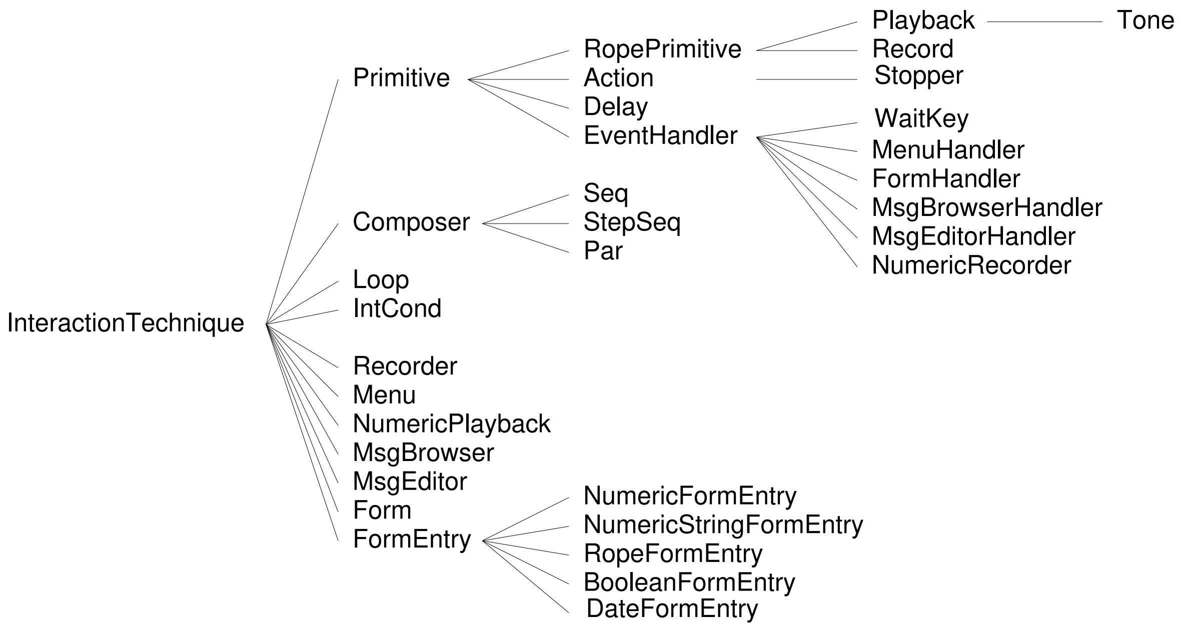

Figure 2: VitKit class hierarchy

5 Interaction techniques

Each class in the VitKit library corresponds to a kind of interaction technique. Instances of interaction techniques can be combined to build user interfaces. Programmers can derive their own application-specific interaction techniques from VitKit classes to build libraries of re-usable components.

Figure 2 illustrates the class hierarchy[3]. Interaction techniques can broadly be classified as primitives, composers, flow controllers, and higher-order techniques.

Primitive interaction techniques provide the basic building blocks for building interfaces. Structuring techniques can be used to combine these building blocks into more complex interaction techniques (also known as higher-order techniques). VitKit also provides a set of general-purpose higher-order interaction techniques which includes vidgets for menus, message browsers, and form filling. Vidgets written by an application programmer will normally be of the higher-order type. However, there is nothing to stop the application programmer from writing additional primitives and composers as long as a few basic conventions are adhered to. The entire voice interface may be viewed as the highest order interaction technique in an application.

The rest of this section describes the toolkit classes in some depth. Following an overview of the operations common to all interaction techniques (the **InteractionTechnique** protocol), the primitives, structuring techniques, and higher-order interaction techniques are discussed in that order. A few examples of their intended usage are included.

[3] In the library, all class names are prefixed with the characters 'vit_' so as to minimise the possibility of name clashes with other software. This is a temporary mesaure which will not be necessary once the new C++ 'namespace' feature becomes available. For clarity reasons, the prefix will not appear when referring to VitKit classes in this text.

```
class InteractionTechnique {
public:
    typedef enum { OK, Stopped, Failed } Result;
    virtual Result Execute(PhoneDev*) = 0;
    virtual Result Stop() = 0;
    virtual Result Pause() = 0;
    virtual Result Resume() = 0;
    virtual Result Wait();
    Result Run(PhoneDev*);
};
```

Figure 3: The **InteractionTechnique** class.

5.1 Interaction Technique Protocol

All interaction technique protocols contain a common set of control operations. Some interaction techniques provide additional operations which are only meaningful to themselves. At the root of the hierarchy lies the abstract class **InteractionTechnique** (figure 3) which establishes the common operations that all interaction techniques must implement.

Execute() commences execution of an interaction technique and returns immediately. **Stop()** cancels this execution. It is safe to call **Stop()** on an interaction technique after its execution has already terminated. **Pause()** and **Resume()** may be used to temporarily suspend and continue an execution. **Wait()** is a synchronisation operation which returns when the execution has terminated (whether naturally, as a result of a call to **Stop()**, due to the caller hanging up, or some other failure). Note that some interaction techniques never stop naturally, but must be stopped explicitly. An example of this is the **Loop** (which repeatedly executes its contents forever).

The **Run()** operation first establishes a communications link with the user terminal (represented by the indicated **PhoneDev** object) and then calls **Execute()** to commence execution of the interface.

5.2 Primitives

There are various kinds of primitives, each representing some basic function that can be utilised by higher-order interaction techniques.

Two primitives are associated with ropes: **Playback** to play pre-recorded ropes, and **Record** to record ropes. **Delay** can be used to generate silence. **EventHandler** is an abstract class, subclasses of which are intended to handle keypad presses. **Action** is also an abstract class, subclasses of which are intended to implement callback routines that may be called at any point from within an interface.

The ability to insert a callback at any point in the interface gives maximum flexibility as to what can be done with the toolkit. The programmer can also freely mix the VitKit interface specification with ordinary C++ code as well as ANSAware DPL statements, enabling remote procedure calls to be made from within VitKit interfaces.

The **Reader** vidget produces synthesized speech from a string of text. It is extremely useful in building user interface prototypes. Synthesized prompts can be used during development, to be replaced later by pre-recorded prompts via **Playback** vidgets. **Reader** is also useful for conveying textual information through an audio connection when no visual display is available. For example, it may be used to have one's electronic mail read over the phone.

Tone is a **Playback** vidget that plays a tone when executed. This is useful as a cue to the user immediately before recording a rope.

Stopper is an **Action** vidget that takes an interaction technique as argument. The effect of executing a stopper is to stop the execution of the passed interaction technique. An example of its use is given in 5.3.1.

WaitKey is an **EventHandler** vidget that simply waits for the specified keys to be pressed before terminating.

Examples

In all ensuing examples it is assumed that **ir_vrs** is a valid rope server interface reference and **pd** is a valid pointer to a **PhoneDev** object.

The following code fragment shows how a **Playback** can be used to play a rope. Note that one should wait for the **Playback** execution to finish before deleting the object.

```
    Playback *p = new Playback(&ir_vrs, ropename);
    p->Run(pd);
    cout << "Playing rope...\n";
    p->Wait();
    delete p;
```

The next example defines a **DumbEventHandler** class which does nothing and terminates when any keypad button is pressed.

```
class DumbEventHandler : public EventHandler {
public:
    void Handle(const Event&);
};

void DumbEventHandler::Handle(const Event &e) {
    switch(e.type()) {
    case Event::Button:
        Stop();
    }
}

void body() {
...
    DumbEventHandler *deh = new DumbEventHandler();
    deh->Run(pd);
    deh->Wait();
    delete deh;
...
}
```

This functionality is actually available directly via the **WaitKey** primitive. The above **DumbEventHandler** is equivalent to the vidget:

```
WaitKey(B0|B1|B2|B3|B4|B5|B6|B7|B8|B9|BStar|BHash)
```

5.3 Structuring techniques

5.3.1 Composers

A composer is an interaction technique that combines an arbitrarily long sequence of sub-techniques (children) in some manner. All composer types are similar in that they maintain a list of their children. This list (which is terminated by a NULL pointer) is normally passed statically to the constructor. However, there are also operations to manipulate the list dynamically; items may be added and removed from the list at run-time.

Composers may differ in their execution semantics. At present three composer classes are defined, namely **Seq** (which executes each of its children in turn sequentially), **StepSeq** (which repeatedly executes the current technique and moves on to the next when explicitly instructed to do so) and **Par** (which executes its children simultaneously).

Examples

The following code plays the ropes whose names are 'one', 'two', and 'three' in that order.

```
Seq *seq = new Seq(
    new Playback(&ir_vrs, "one"),
    new Playback(&ir_vrs, "two"),
    new Playback(&ir_vrs, "three"),
    NULL
);
seq->Run(pd);
seq->Wait();
delete seq;
```

The next example demonstrates how a stopper can be used in a composition to stop a playback prematurely.

```
Playback *p = new Playback(
    &ir_vrs, ropename
);

Seq *s = new Seq(
    new WaitKey(Event::B0),
    new Stopper(p),    // Stops playback
    NULL
);

Par *par = new Par(
    new Seq(
        p,
```

```
            new Stopper(s),    // Stops WaitKey
            NULL
        ),
        s,
        NULL
    );

    par->Run(pd);
    ...
```

Note the two uses of the **Stopper** vidgets. One is used to terminate execution of the **Playback** if button 0 is pressed. The other is used to terminate the event handling composition **s** if the execution of the **Playback** terminates naturally.

The **SkipForward()** and **SkipBackward()** members of the **Seq** composer can be useful for constructing skip-and-scan type interfaces as described by Resnick [RV92][4]. The following example allows skipping back and forth through a list of **Playback** objects using the 7 and 9 keys. The session is terminated upon hitting 0, or upon completion of execution of the last **Playback** in the sequence.

```
class DemoEventHandler : public EventHandler {
public:
    DemoEventHandler(Seq *i) : it(i) { }
    virtual void Handle(const Event&);
private:
    Seq *it;
};

void DemoEventHandler::Handle(const Event &e) {
    switch(e.type()) {
    case Event::Button:
        switch (e.buttonValue()) {
        case Event::B0:
            it->Stop();
            break;
        case Event::B7:
            it->SkipBackward();
            break;
        case Event::B9:
            it->SkipForward();
            break;
        }
    }
}
```

[4] Resnick shows that it is possible to create rather sophisticated, yet usable, user interfaces based on a pair of input and output audio channels and a numeric keypad. This may be achieved by the use of techniques such as skip-and-scan forms and menus. In VitKit, skip & scan is viewed as a feature of some widgets rather than an overall interface style.

```
void body() {
    ...
    Seq *seq_p = new Seq(
        new Playback(&ir_vrs, "long1"),
        new Playback(&ir_vrs, "long2"),
        new Playback(&ir_vrs, "long3"),
        NULL
    );

    DemoEventHandler *deh_p = new DemoEventHandler(seq_p);
    seq_p->Append(new Stopper(deh_p));

    Par *par = new Par(
        seq_p,
        new DemoEventHandler(seq_p),
        NULL
    );
    par->Run(&ir_pd);
    par->Wait();
    delete par;
    ...
}
```

5.3.2 Other structuring techniques

The remaining structuring techniques deal with different forms of sequential flow control. The **Loop** technique can be used to repeat execution of a vidget until expicitly stopped. The **IntCond** technique is a multi-conditional structure (analogous to the switch statement in C). Both these techniques are used in the implementation of some of the higher-order interaction techniques discussed below.

5.4 Higher-order interaction techniques

Higher-order interaction techniques use structuring techniques to combine primitive vidgets into more useful interface components. This section discusses some issues related to higher-order interaction techniques and then describes two particular techniques, namely the **Menu** and **Form** vidgets.

Session

Higher-order vidgets may need to make use of some shared state data. This is captured by means of a **Session** object, which is initialised prior to execution of an interface. There can only be one instance of this object (per execution environment) at any one time.

One use for the session object is in higher-order techniques which need to make use of pre-recorded audio ropes. For example, the **NumericPlayback** vidget makes use of a set of 10 ropes, each corresponding to a different spoken digit. Such ropes are regarded as part of the toolkit and are stored separately from application ropes using a different rope server. VitKit techniques making use of such ropes obtain an interface reference to this server from the **Session** object.

Help

The more sophisticated techniques may be harder for users to become accustomed to. Spoken help messages are therefore provided for most higher-order vidgets and are accessible via the telephone keypad hash key.

The Menu vidget

This vidget repeatedly reads out a list of option prompt vidgets (supplied as arguments) until a numeric key is pressed. The star key may be used to skip forward quickly to the next item. Pressing a numeric key results in an associated vidget (typically an action) being executed. Following execution of this vidget, execution of the menu terminates.

A menu which is returned to after execution of one of its options can be implemented by placing the menu in a loop.

The Form vidget

A form consists of a number of fields, each of which corresponds to a value of some type. Supported types include voice, numeric strings, and dates.

At any time during the execution of a form, exactly one of its fields is said to hold the focus. The title and value of the field holding the focus is repeatedly read out. The 7 and 9 keys on the pad can be used to shift the focus forwards and backwards through the form. The 1 key can be used to record new values for the current field; pressing this will cause the system to go into record mode and the user will be prompted to speak or key in a value, depending on the field type. The 3 key can be used to erase the value in the current field. The 0 key is used to exit the form upon completion.

6 The ODO Voice Messaging Service

As an illustration, this section describes how VitKit is used in the the ODO voice message service. Rather than record a message as one continuous stream, a form is used to record a semi-structured message (the general benefits of which are discussed in [MGL+87]). For example, the name of the sender and the subject of the message might be recorded as separate fields so that these can be incorporated into a selection menu when the receiver accesses his voice mailbox.

The structure of a message is determined by a template. There are a number of standard templates but it is also possible for users to define their own.

A template definition consists of a list of field names and types, together with a corresponding list of prompts and headings to use for input and output and an indication of which fields are compulsory and which fields are to be played back when a user is browsing his list of messages. For example, the standard template for plain messages consists of three voice fields: name, subject, and message. All fields are compulsory and the name and subject are designated as selection fields.

A template might also specify a number of processing options. These are made available to the recipient upon selection of the message. Each option consists of a voice prompt and an action specification. The latter is written using the ODO call management specification language [RLU94a].

One example of a processing option can be found in another standard template for 'get-back-to-me-on-this-number' requests. This template uses a numeric field to allow the caller to leave a phone number. A processing option is specified which enables the recipient to call back the caller using the specified number.

A user might set up a user-defined template for some specific purpose eg collecting names, particulars and information requests from people wanting to know more about a particular product.

6.1 Components of the ODO voice message service

The service makes use of three kinds of server: one that stores message descriptors (VMsgDescStorage), another that stores user mailboxes (VMsgMailboxStorage), and of course the rope server for storing audio field values. There may be more than one instance of each of these.

A user mailbox simply consists of a list of references (actually capabilities[5]) to message descriptors (which are not all necessarily stored in the same VMsgDescStorage server).

A message descriptor contains information about the message (such as the date, time and source) and values of non-audio fields. Additionally it contains an interface reference for the rope server where values of voice fields are stored, together with details of the ropes themselves. Every message also has its own unique reference number (generated via an operation in the VMsgDescStorage interface). Last but not least, the descriptor also contains a reference to the corresponding message template.

Two voice interfaces were constructed for the message service, one for leaving messages (sender), the other for reading messages (reader).

6.2 The Sender interface

When a user successfully connects to the message sender interface he is greeted with an introductory message followed by a menu offering a number of different message templates. The list of templates offered depends on the callee's policy. If only one template is available, the menu is skipped.

The caller then fills in the selected message template by means of a dynamically-created form vidget. In order to enable re-usability of interface components, the VMsgDescStorage server generates a unique message identifier which is incorporated in all rope-names corresponding to voice fields in the message. This ensures that existing ropes are not overwritten accidentally. Upon completion of the form, a transaction is executed to create the message descriptor and post it in the appropriate mailbox.

6.3 The Reader interface

Users access their messages via the reader interface. Upon connection, the caller engages in a selection process to select a message. Once this has been done, a processing menu is presented,

[5] A capability is a reference embodying access control information. A good introduction to capabilities in distributed systems can be found in [Mul89].

minimally offering the options to listen to the selected message and to delete it. Depending on the message type, further options may be offered as specified by the corresponding template. For example, if the message contains a telephone number to call back, an option might be provided to make the call using the specified number as described earlier. In general, the selected message can be passed on to some other service (specified in the template) for processing. The interface eventually returns to the selection phase in order to deal with the next message.

The selection part of the interface uses a dynamically constructed **StepSeq** composer together with an event handler similar to that given in the last example of section 5.3.1. The selection fields of the current message are played repeatedly whilst the keypad is used to navigate through the message list and make a selection.

Deletion of a selected message is immediately reflected in the interface by removing the appropriate vidget from the **StepSeq** composition. Additionally, any new messages are incorporated into the **StepSeq** before re-entering the selection phase.

The processing menu always minimally contains options for playback and deletion. Additional processing options specified by the template are added dynamically.

7 Discussion

In this paper, the VitKit toolkit for telephone-based interfaces was described. The toolkit consists of a library of interaction techniques that can be used to build sophisticated user interfaces. These interfaces can interact with distributed applications and can also be constructed and modified dynamically at run-time.

VitKit is fairly low-level and is harder to use than the higher-level tools described earlier because it requires knowledge of programming in C++. Ease-of-use was sacrificed for flexibility, but this was born out of necessity. None of the tools mentioned earlier would have been suitable for the kind of interfaces required in the ODO project.

To date, VitKit has only been used in the context of the ODO project, but there is nothing to stop it being used in other applications. In many cases, higher-level tools might be preferred, especially where static interfaces requiring little integration with other applications are sufficient. However, if more flexibility is required then the VitKit approach is clearly essential. Even though programming is required, the object-oriented design makes it relatively easy to understand programs, to combine and re-use components, and to build libraries of re-usable components. In this respect, the approach enjoys the same benefits as do analogous toolkits in the graphical interface world.

Currently VitKit assumes a basic telephone terminal providing bi-directional voice and a numeric keypad. However, recent years have witnessed the emergence of more sophisticated terminals providing graphics and video capabilities as well as more sophisticated keypads. Telephones can also be paired with workstations to provide graphical interface support. One interesting possibility we are pursuing involves extending the toolkit to automatically support text-based or graphical-based user interaction whenever an appropriate terminal is available [6]. This would involve separating the presentation of a vidget from its purpose, enabling several different presentation types for the same vidget.

Other planned future enhancements to VitKit include incorporation of rate control for **Playback** vidgets (enabling fast-forwarding and rewinding through long messages), the addition of

[6]This mirrors the directions taken by the InterViews toolkit in its evolution from version 2.6 to 3.1.

a **Recogniser** primitive (for recognition of spoken commands), and a number of other general-purpose higher-level interaction techniques. It is envisaged that as we gain more experience building user interfaces with VitKit, the class hierarchy will continue to evolve.

References

[Arc92] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, United Kingdom (apm@ansa.co.uk). *RM.099.02: An Overview of ANSAware 4.1*, May 1992.

[G+87] John D. Gould et al. The 1984 olympic message system: a test of behavioral principles of system design. *Communications of the ACM*, 30(9):758–769, September 1987.

[GB83] John D. Gould and Stephen J. Boies. Human factors challenges in creating a principal support office system—the speech filing system approach. *ACM Transactions on Office Information Systems*, 1(4):273–298, October 1983.

[Int] International Standards Organisation. *ISO/IEC 10746 (ITU-T Recs X.901-904) Reference Model for Open Distributed Processing.*

[Li94] Ning Li. A distributed audio system. In *Multimedia/Hypermedia in Open Distributed Environments*, pages 109–121. Springer-Verlag, Wien, Austria, 1994.

[Lin92] Peter F. Linington. Introduction to the basic reference model of open distributed processing. *IFIP Transactions C, Special Issue on Open Distributed Processing*, C-1:3–13, 1992.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[MGL+87] Thomas W. Malone, Kenneth R. Grant, Kum-Yew Lai, Romana Rao, and David Rosenblit. Semistructured messages are surprisingly useful for computer-supported coordination. *ACM Transactions on Office Information Systems*, 5(2):115–131, April 1987.

[Mul89] S.J. Mullender. Protection. In Sape Mullender, editor, *Distributed Systems: Concepts and Design*, chapter 7. Addison-Wesley, 1989.

[RBG86] John T. Richards, Stephen J. Boies, and John D. Gould. Rapid prototyping and system development: Examination of an interface toolkit for voice and telephony applications. In *CHI '86 Conference Proceedings*, pages 216–221. ACM, New York, April 1986.

[Rep92] Alex Repenning. Using agentsheets to create a voice dialog design environment. In *Symposium on Applied Computing, Kansas City*, pages 1199–1207. ACM, New York, 1992.

[Res92] Paul Resnick. Hypervoice: A phone-based cscw platform. In *CSCW '92 Conference Proceedings*, pages 218–225. ACM, New York, November 1992.

[Res93] Paul Resnick. Phone-based CSCW: Tools and trials. *ACM Transactions on Information Systems*, 11(4):401–424, October 1993.

[RLU94a] Mike Rizzo, Peter F. Linington, and Ian Utting. Call management in the Open Distributed Office. Technical Report 15-94, Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent CT2 7NF, United Kingdom, August 1994.

[RLU94b] Mike Rizzo, Peter F. Linington, and Ian Utting. The ODO project: a case study in integration of multimedia services. Technical Report 12-94, Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent CT2 7NF, United Kingdom, August 1994.

[RV92] Paul Resnick and Robert A. Virzi. Skip and scan: Cleaning up telephone interfaces. In *Human Factors in Computing Systems: CHI '92 Conference Proceedings*, pages 419–426. ACM, New York, May 1992.

[Sch93] Chris Schmandt. Phoneshell: the telephone as computer terminal. In *Proceedings of Multimedia '93*, pages 373–382. ACM, New York, 1993.