66

DynaDesigner: A Tool for Rapid Creation of Device-Independent Interactive Services

Loren Terveen and Mark Tuomenoksa
AT&T Bell Laboratories
{terveen, mlt}@research.att.com

KEYWORDS service creation tools, device-independent design, dialogue design, end user programming, consumer systems

ABSTRACT DynaDesigner is a tool for creating, testing, and deploying interactive services to be delivered on devices such as telephones, TVs, and PCs. A key feature is that it supports *device-independent* service design — a service is designed once, independent of any particular device. This eases the design and maintenance task for service providers and makes services easier for consumers to use, since they are consistent across devices. DynaDesigner has been used to design and deploy many services. With DynaDesigner, services can be designed and deployed in hours.

INTRODUCTION

The convergence of computers, communications, and consumer electronics brings a new urgency to the problem of creating easy to use systems — now all people with telephones and TVs, not just PC owners, are potential users. Further, the proliferation of devices raises the challenge of creating *device-independent* services — consumers may want to use a service such as home banking on different devices such as telephones, TVs, PCs, or PDAs — and they will want consistency across devices. We have responded to these challenges by creating a service creation tool called *DynaDesigner*. Direct manipulation, form-based editing, and graphical simulation are used to design and test services. Code is automatically generated to implement services on standard hosting platforms. Device-specific interpreters enable the same service to be delivered on many devices and guarantee a consistent, high quality interface for each device.

TRANSACTIONAL SERVICES

DynaDesigner targets the class of transactional services that includes activities such as home banking, bill paying, ticket purchasing, and catalog shopping. In such services, a consumer completes a transaction by reading or listening to information, selecting options from menus, and supplying information such as personal identification numbers.

This type of service typically is implemented in a distributed, network-based architecture. A program that implements the service logic runs on a *hosting platform* — a computer on the service provider's premises or in the telephone network that is equipped with special purpose voice and data transmission hardware. The service provider's databases typically are on a large mainframe computer. And the service must be delivered on a consumer device such as a telephone, TV, or PC. The hosting platform, database machine, and delivery devices are connected by a combination of telephone lines and high speed data links.

DYNADESIGNER

DynaDesigner is a service creation tool that addresses the following goals:

- *Service creation should be fast and easy.* Competitive pressures make speed necessary; speed requires an easy to use system.
- *Services created should be of high-quality.* Making service creation fast and easy is of no use unless consumers want to use the services.

K. Nordby et al. (eds.), *Human-Computer Interaction*
© IFIP International Federation for Information Processing 1995

- *Services should be device-independent.* A device-independent representation of services aids consumers (i.e., service users) by ensuring service consistency across devices and aids service providers since they have just one service to design and maintain.

DynaDesigner achieves these goals with two major features:

- Service authoring consists of specifying the service logic and information content, rather than designing an interface. DynaDesigner uses interface "templates" to produce an appropriate interface for each delivery device from the same logic and information content. This helps service providers by making service design and modification faster and easier. It helps end users by guaranteeing a consistent, high-quality interface across services and devices. And it is the key enabler of device independent services.
- DynaDesigner supports *end-to-end* service creation, i.e., both design and implementation via automatic code generation. This is necessary to make service deployment and modification fast.

Computationally, the type of services we are concerned with can be modeled naturally as state machines. However, DynaDesigner is not a general purpose visual state machine programming system. Rather, it offers service designers a set of domain-specific (but *device-generic*), high-level abstractions for creating a service. A domain-specific [Fischer 1992] approach makes service creation much easier, enabling non-programmers to build services.

Service creation building blocks include *computational* (or *logic*) objects, such as objects for querying and updating databases, branching based on data comparisons or on the time, date, or day of week, and transferring calls to customer service representatives. Clearly, the logic of a service is device-independent — the challenge is coming up with a device-independent way of specifying interactions with a user. We do this by providing a set of generic *interaction* (or *dialogue*) objects that encapsulate common types of communications with users — such as *Present Information*, *Get Input*, and *Present Options* — but do not specify the surface details of how the communication should be carried out.

A designer specifies the logic of a service by selecting objects from a palette, positioning them on a design pad, then linking the objects. For example, a home banking application might begin with a *Present Information* object that welcomes the user, continue with a *Get Input* object that asks the user to enter a personal identification number (PIN), then a *Database Lookup* object to verify that the PIN is valid, and next contain a *Present Options* object that lets the user select from a set of banking services.

Each type of object encapsulates "natural" (as shown by our experience in working with service providers) chunks of information and user response types. For example, each of the interaction objects has Title, Information, and Instruction fields, and the *Get Input* object includes standard actions that allow an end user to signal that input is complete or to erase input, so the designer does not have to remember to add these actions. Each type of object has one or more editors for filling in the information required for that object. For example, each interaction object has editors for specifying the information to present to users and possible user responses (in addition to the default responses built into the object). The *Get Input* editor also lets the designer specify a variable for saving the user input (thus making it available to the rest of the service) and select one of several common ways to validate the user input, e.g., by looking it up in a database table. More complicated validation can be performed with other objects such as database lookups and data comparisons, but the *Get Input* object makes it very simple to do the most common types of validation.

Figures 1 and 2 show the editors for specifying the information content and user actions for a *Get Input* object that collects a user's PIN. They help show how we maintain device-independence. Clearly the Title ("Enter PIN") and Information ("Please enter your personal identification number") text is device-independent (i.e., it can be spoken or displayed). However, the Instruction field, which tells users how to proceed, is a bit more complicated. The instruction "Press [CONT] when you are finished or [BKSP] to erase a character" contains references to two device-independent actions, [CONT] (for "CONTinue") and [BKSP] (for "BacKSPace"). An *action editor* lets a designer assign a device-specific

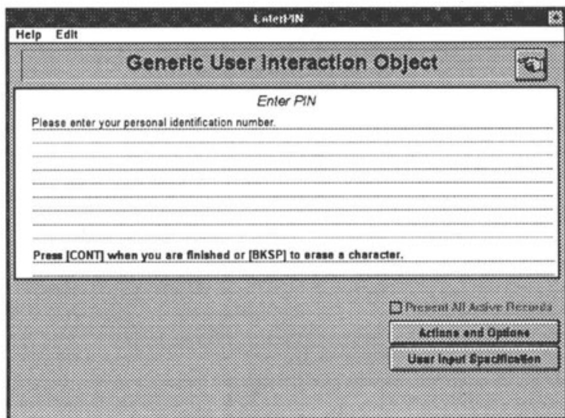

Figure 1: Editor for specifying information content for an interaction object

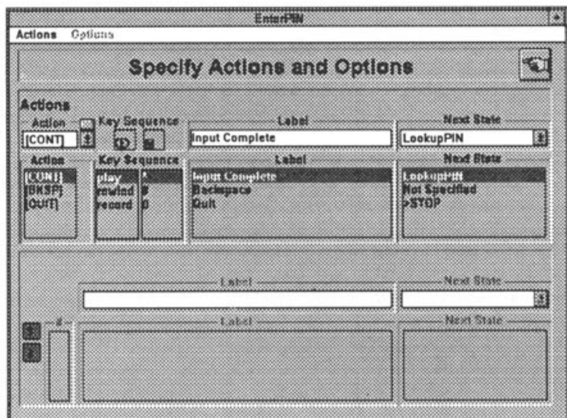

Figure 2: Editor for specifying actions users may take in response to an interaction object

key that a consumer uses to signal the action (e.g., "*" might be used on a touchtone phone) and a short phrase for describing the action (e.g., "Continue" or "Next" for [CONT]). In addition to device-independence, this ensures intra-service consistency: action definitions are global to a service, so it is impossible for a designer to use (for example) the "*" key for [CONT] in one state and the "# key in another.

This illustrates another aspect of our approach: we keep the *core* of the service specification device-independent, but provide principled ways to specify and use device-specific information. First, a device-specific interface template for each interaction object is used to generate the speech or display and manage the user input for that object. For example, suppose a *Menu* object contains three options, say "Account Balance", "Transfer of Funds", and "Order New Checks". The template for spoken menus and text-to-speech synthesis would generate the spoken instructions "For Account Balance, press 1", "For Transfer...". Users could select an option by pressing the appropriate key on the touchtone pad. The template for TV display screens would result in each item being presented on a separate, numbered line. Users could select an option by pressing the appropriate button on a universal remote control.

Second, editors like the action editor collect device-specific information. While the action-key mappings gathered by this editor are required, other editors capture optional annotations which enhance the presentation of a service on a particular device. For example, a *phrase manager* for spoken output allows text to be broken up into phrases, each of which can be spoken using a professionally recorded voice, and an *appearance editor* for screen output allows designers to specify bitmaps to use as screen backgrounds and tailor display properties such as font, background color, etc.

At any time, a service designer may simulate a service, determining whether the logic is correct and getting a feel for what the interaction would be like on a specific device. Graphical simulation, single-stepping, and stop-points make it easy to test and debug a service.

When a service is complete, DynaDesigner automatically generate the code to implement the service on a standard service hosting platform. Currently, code is generated for the Conversant™, a special purpose AT&T computer that resides in the AT&T network or on the premises of a service provider. It can send and receive voice or data over telephone lines and can communicate with other computers as necessary, for example, to access a database.

We have developed DynaDesigner in close collaboration with service providers, building many services to ensure ease of use and broad coverage. DynaDesigner went into production use in August 1994. About 50 voice services have been deployed.

Using DynaDesigner resulted in dramatic decreases in the time it takes to design and deploy a service. The application of DynaDesigner to interactive TV services is newer, and it was this application that led us to develop the device-independent interaction objects. To date, we have designed several services based on these new objects and will be implementing and deploying these and other such services in the near future.

RELATED WORK

DynaDesigner is related to cross-platform GUI construction tools and end user programming environments. Like a GUI builder, it enables the design of an interactive system that can run on different devices. However, with DynaDesigner, designers specify dialogue structure and content, not widgets and layout. And rather than delivery across different windowing systems or interface toolkits, DynaDesigner services are generic across a range of devices, from televisions to telephones. As an end user programming system, DynaDesigner enables users who are not programmers to create applications. Unlike general purpose visual programming (Glinert 1986) or state machine (Carneiro et al 1994, Wellner 1989) systems, DynaDesigner is a domain-specific system. It provides a set of high-level, domain-specific abstractions for building applications. Other systems exist for creating voice dialogues, both commercial tools such as TFLX, VOS, and Visual Voice, and research prototypes such as VDDE (Repenning & Sumner 1992). Our work is distinguished from all of these systems by our provision of high-level, device-independent interaction objects. Further, we automatically generate the code to implement services on standard network-based, hosting platforms, which typically means that many more customers can be serviced.

FUTURE WORK

First, we are refining the DynaDesigner objects in response to user feedback. For example, we have created objects for dealing with databases that are simpler to use, encapsulate both data access and user interaction, and which are both more cleanly device-independent and better exploit the unique resources of voice and display devices. Third, we are beginning to explore the creation of services that use speech for input and produce coordinated speech and visual output.

CONCLUSIONS

We conclude by summarizing lessons this work has taught us. First, by providing high-level abstractions for creating services, we both ease the task of service creation and enable device-independent services. Service creation requires neither programming nor interface design, allowing service authors to concentrate on the content of a service. Second, end-to-end service authoring requires addressing infrastructure issues; thus, we generate code to implement services on standard hosting platforms, to handle distribution of computation in network-based services, and to interface to legacy databases. The combination of high-level building blocks and end-to-end service generation makes DynaDesigner a very usable and useful tool.

REFERENCES

Carneiro, L.M.F., Cowan, D.D., and Lucena, C.J.P. (1992) ADVCharts: A Visual Formalism for Interactive Systems, *SIGCHI Bulletin*, 1992, Vol. 26 No.2, pp. 74-77.

Fischer, G. (1992) Domain-Oriented Design Environments, In *Proc. Knowledge Based Software Engineering Conf.*, 1992, IEEE Press, pp. 204-213.

Glinert, E.P. (1986) Towards ``Second Generation" Interactive, Graphical Programming Environments, In *Proc. IEEE Comp. Society Workshop on Visual Languages*, 1986, IEEE Press, pp. 61-70.

Repenning, A. and Sumner, T. (1992) Using Agentsheets to Create a Voice Dialog Design Environment, In *Proc. ACM Symposium on Applied Computing*, 1992, ACM Press, pp. 1199-1207.

Wellner, P.D. (1989) Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation, In *Proc. CHI'89*, 1989, ACM Press, pp. 177-182.