Flexible, Dynamic User Interfaces for Web-Delivered Training

Srdjan Kovacevic
U S WEST Advanced Technologies
4001 Discovery Drive, Boulder CO 80303 USA
Tel: +1-303-541-6381
E-mail: srdjan@advtech.uswest.com

ABSTRACT

One of the critical parts of a tutoring system is its user interface (UI), which must neither constrain an author in developing lessons, nor impede a student during practice. A system providing training over the Web must also address issues of interface transport, providing feedback and managing local context. We have developed a system, MUSE, that applies a model-based technology to address the above requirements. It supports a wide range of interface styles. Resulting UIs can be customized and capture enough application semantics to provide local feedback and manage the context required for evaluating a student's work and providing coaching.

Keywords

UI models, UI design tools, UI components, UI representation, model-based design, intelligent tutoring system, Web interfaces, Web-delivered training, application semantics.

INTRODUCTION

A new generation of tutoring systems, supporting Web-delivered training, places requirements on user interface (UI) development that go beyond those of traditional tutoring systems. A traditional tutoring system requires a UI framework that will give a designer enough flexibility to convey the material taught in the least obtrusive way, allowing both a designer (at design time) and a student (at run time) to focus on the content of each lesson. It requires a powerful design environment that supports a wide range of interaction styles and easy exploration of the UI design space, so that an instructor developing training material is not constrained by the lack of UI capabilities and forced into adopting an inadequate teaching paradigm. The UI framework should also allow users (students) to customize UIs according to their preferences.

Web-delivery, where students can access a tutor on a remote system over the Web, through a local UI client, brings additional requirements. The course material should not be hard-wired into the UI clients, as that would make it very difficult to maintain and update the content. A framework in which a server controls UI definitions for each lesson and sends them to clients when needed must support interface transport between the server and clients. Yet, transporting a simple UI is not enough. Because a UI client interacting with a student is physically separated from the server doing evaluation and providing coaching, the client cannot rely on the server to provide semantic feedback. Rather, the UI client must be able to handle semantic feedback locally, as well as maintain the context during the session, with all relevant information that the coach may need for evaluating the student's work. This means that the UI client must capture enough semantics locally and be able to maintain the whole session (between requests for coach evaluation), without having to communicate with the server for each user interaction.

We have developed the SLOOP intelligent tutoring system as a framework for Web-based training delivery using the model-based approach [Arens et al. 1991, Sukaviriya and Kovacevic 1994] to circumvent the limitations of traditional UI tools and HTML browsers. SLOOP uses an HTML browser (currently Netscape™) for what they are good at – displaying static information – but for UIs supporting dynamic interaction, SLOOP uses MUSE (Model-based User interface for SLOOP Environment). In the SLOOP framework, the MUSE server maintains UI descriptions and transports them as high-level specifications to MUSE clients which then instantiate a UI based on the information contained in the specification. The resulting UI captures the application semantics needed for providing semantic feedback without having to communicate with the server for each user interaction.

In this paper we describe MUSE using a detailed example. The paper is organized as follows. Section 2 reviews related work and problems with traditional UI tools and HTML browsers. Section 3 gives an overview of the SLOOP architecture. Section 4 discusses UI development and management using MUSE. It begins with a high-level overview of the UI development process. Next it describes the content of the high-level specification used for interface transport and generation, the underlying application conceptual model and UI model, and the steps in the UI generation process. Section 5 revisits the examples to point out features of UIs generated in MUSE that are not supported by HTML browsers. Section 6 describes the implementation, and Section 7 offers conclusions and directions for future work.

RELATED WORK

In order to support the construction of UIs that satisfy the requirements discussed above, the underlying UI framework must have the capability to produce and deliver dynamic, flexible UIs; to support a wide range of interaction styles, including graphics and direct manipulation; to support customizable UIs;

AVI '96, Gubbio Italy
© 1996 ACM 0-89791-834-7/96/05..$3.50

and to transport and dynamically load interfaces that capture enough application semantics to maintain sessions and give adequate feedback.

Traditional interface development tools do not meet these requirements. Interface builders are easy to use, but focus on low level details and do not provide adequate support for exploring designs and building customizable UIs [Szekely et al. 1993]. Furthermore, they do not support interface transport, which requires a framework in which a UI client can provide an interface from a description sent over the network.

Currently, UI tools do not explicitly provide such a framework. Tools supporting automatic generation of UIs from high-level specifications provide enabling technology for interface transport. However, most of those tools focus on menus and dialog boxes (e.g., Mickey [Olsen 1989], Chisel [Singh and M. 1989], ITS [Wiecha et al. 1990], DON [Kim and Foley 1993], TRIDENT [Vanderdonckt and Bodart 1993]), thus limiting the range of possible UIs. Also, most of the tools supporting automatic generation typically translate the specification into a form requiring compilation (e.g., MIKE [Olsen 1986], ADEPT [Johnson et al. 1993]) and are thus not suitable for dynamic generation at run time.

HUMANOID [Szekely et al. 1992] and UIDE [Foley et al. 1991, Sukaviriya et al. 1993]) use more sophisticated models, which are interpreted at run time and support a wider range of interaction styles. MASTERMIND [Neches et al. 1993, Szekely et al. 1995] is an effort to build a comprehensive model-based environment by integrating the HUMANOID and UIDE models and building on their strengths; it is still in the design stage, but is expected to provide both compiled and interpreted run-time UI support.

MUSE, presented in this paper, is based on the TACTICS model [Kovacevic 1992a, Kovacevic 1992b, Kovacevic 1994], which is also derived from UIDE. TACTICS uses an application conceptual model comparable to HUMANOID and UIDE, but it also has an explicit UI model and transformations for mapping the application conceptual model into the UI model, as well as for transforming UIs into a desired look and feel. TACTICS supports dynamic generation because it does not require compilation of UI structures. In addition, it instantiates the run-time UI structure which then executes without interpreting the application model, thus minimizing the run-time overhead compared to the interpreted environments such as HUMANOID and UIDE.

Standard Generalized Markup Language (SGML) standards [Newcomb et al. 1991] provide a framework for interface transport based on a document paradigm. An SGML specification, instead of describing formatting features directly, describes document structure which a display engine can map to presentations. SGML standards have enabled development of HTML (Hyper Text Markup Language) display engines, or browsers, such as Mosaic and Netscape. Their growing popularity is due to their ease of use, simplicity, and effectiveness in presenting information [Laufmann 1994].

Capabilities of user interface tools depend on their underlying model, how expressive it is and how much application semantics it captures. For instance, Web browsers also use a model, which is what enabled their success, but theirs is a very simple model, that of a hypertext document. Thus, these browsers are limited to document browsing and forms-based display and input. Graphical (direct) manipulation UIs are not supported, because the browsers do not have the notion of objects to be manipulated with semantics behind the manipulations.

These HTML browsers also lack the notion of a session and a context history. If the order of interaction steps matters, the interaction must be broken into a sequence of documents. Similarly, a UI in which what is available/enabled/presented to a user depends on a previous context is not directly supported but requires additional helper applications acting as filters and session managers. Such a UI has to be captured in a set of documents, either predefined or dynamically generated. For instance, updating a list of items based on the most recent selection requires fetching a new page and using either helper applications that can modify pages or separate, predefined pages for each possible selection.

HTML browsers share some similarities with early User Interface Management Systems (UIMSs), which were limited in terms of the feedback they could provide because of their lack of knowledge about application semantics [Hayes et al. 1985, Tanner and Buxton 1983]. Similarly, HTML browsers are limited to knowledge about document structure and hypertext links. Because in the case of HTML browsers the separation is not only conceptual but also physical, the internal interface may become a real bottleneck, either toward the server providing HTML pages that contain feedback for user selections, or toward helper applications managing semantics needed for providing the feedback.

An additional limitation of HTML browsers is that they do not allow dynamic changes to page contents; i.e., a whole page must be replaced for any change. Some proposed extensions to HTML (e.g., the push and pull mechanism by Netscape [Netscape 1995]) allow limited changes by helper applications. HotJava [Sun 1995] goes a step further, by allowing integrated helper applications ("applets") that can also be shipped around as documents. HotJava is "programmable" and can support graphical interaction and allow packaging more functionality in a UI (not just browsing). However, it does not directly support dynamic creation of UIs, which is one of the fundamental requirements we have. On the other hand, the Java language can provide a delivery platform for such a framework and we are exploring using Java in MUSE.

SLOOP

SLOOP (the System for Learning Object-Oriented Paradigms) is an intelligent tutoring system. Currently, SLOOP provides training for requirements modeling. Future SLOOP modules will address analysis and design modeling.

SLOOP provides an intelligent coaching environment for practicing object-oriented modeling skills. Students perform a series of activities, or steps, to develop a requirements model (implementation independent) of a proposed system, starting from a problem set that consists of a high-level orientation diagram and use case scenarios describing structural and dy-

namic information and exceptions [Hurley and Hughes 1994]. At each step, SLOOP evaluates student responses and provides feedback and different kinds of help.

SLOOP Architecture

The early SLOOP prototype developed in 1994 was a monolithic application, built in Kappa [Intellicorp 1993]. Based on experience with this prototype and with a related intelligent tutoring system, LEAP [Bloom et al. 1995], which is in a deployment phase, we have decided to redesign SLOOP into a client/server architecture with a long term goal of developing a shell for intelligent tutoring systems.

In developing SLOOP, one of our goals was to leverage on existing, already available components; hence our decision to use hyper-text transfer protocols (HTTP) and HTML browsers. However, because of limitations of the current HTML-based technology, we developed MUSE as an additional component for handling interactions not supported by HTML browsers.

Figure 1 shows a simplified view of the SLOOP architecture. We have abstracted details not pertinent to the focus of this paper. For instance, the coach component of the SLOOP server encapsulates representations of expert, instructional and student knowledge, as well as a session manager that keeps track of multiple active sessions and provides necessary information for generating UI requests sent to the UI server. The UI server prepares messages for the UI client, specifying what UI to load, or how to change it. Depending on its interaction requirements, the UI can be loaded as an HTML page handled by the HTML browser, or as a specification handled by MUSE. In both cases, the specification is assembled dynamically, using a predefined template and the current context data (e.g., information specific to a student and a lesson being practiced).

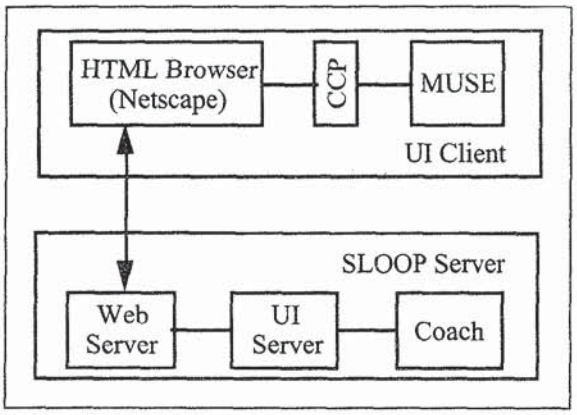

Figure 1 - SLOOP Architecture

The Web server manages the actual communications between the SLOOP server and UI clients, and it consists of an HTTP demon and additional communication programs. The UI client consists of a commercially available HTML browser (currently Netscape), MUSE, and a client communication program (CCP) that links the two.

Student requests are passed from the UI client to the SLOOP server, together with the relevant context. The context can be as simple as a token identifying the type of request (e.g., requests originating from a browser), or as complex as the application context corresponding to the activity practiced. The UI server parses the client requests and propagates necessary information to the coach.

Discussion of the HTML code generation and the relationship between MUSE and the HTML browser is beyond the scope of this paper. In the rest of the paper, we focus on how MUSE handles UI specifications and generates UIs, and what underlying model it uses.

MUSE - UI DEVELOPMENT AND MANAGEMENT

MUSE is based on the TACTICS model, which supports the UI development process shown in Figure 2. The bolded components in the figure pertain to activities performed at run time by MUSE and are discussed below in more detail. An interactive authoring tool can be used to create an initial specification, as well as to modify (transform) the application and its UI (while preserving its functionality [Foley et al. 1991, Kovacevic 1992b]), with changes being propagated back to the specification. The authoring tool and links that are shaded are beyond the scope of this paper.

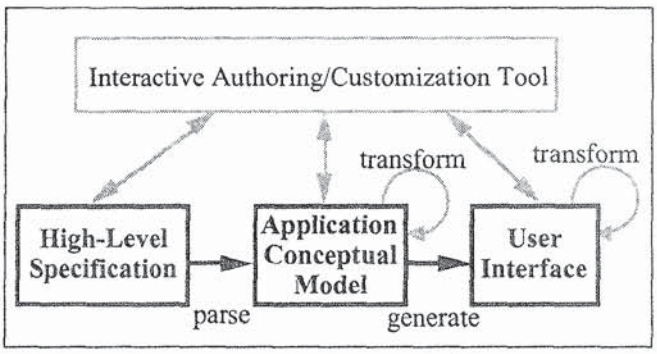

Figure 2 – UI development in MUSE.

A UI (see Figure 2) is generated from a high-level specification that describes application semantics and, optionally, details of a desired look and feel. The high-level specification is first parsed into an application conceptual model. Details of the desired look and feel are associated with the corresponding components of the application model as UI hints. The application model defines application information requirements and a UI is generated to meet these requirements. The resulting UI also tries to achieve the desired look and feel, when possible, by following the UI hints given. The generation process is described in more details using an example based on the SLOOP practice activity *Identify Static Associations*. It is a step in the methodology for building an OO model involving identifying static relationships among entities. The real activity and its UI are more complex, but we have simplified the example in order to better focus on salient points of the generation process. The specification for the activity is shown in Figure 3, and the resulting UI in Figure 4. We now discuss how this graphical, direct manipulation interface is produced from the given specification.

High-level Specification

The specification typically consists of four parts describing the initial context (after ":init" keyword), data types (after ":data" keyword), data model (after ":objects" keyword), and control model (after ":actions" keyword). The initial context is speci-

```
(:application-header StaticAssociations
 :ui (:po ((DialogBox :name "Identify Static Associations" :template Graphical :menubar t)))
 :init (:objects ((noun-phrase (name consortium) ... (name banks) ...))
 :data ((:data-type cardinality :enum (0 Many)))
 :objects ((:object noun-phrase :attributes ((:attr name :type :string)) :ui (:po (nil)))
          (:object actor :ui (:po (Actor))
              :attributes ((:attr name) (:attr cardinality :type cardinality)
                           (:attr external-properties :set-min 0 :set-max nil
                                :type (:object-instance external-property))
                           (:attr noun-phrase :type (:object-instance noun-phrase))))
          (:object object-class ...)
          (:object external-property :attributes ((:attr name)
                                                  (:attr origin :type (:object-instance (actor object-class)))
                                                  (:attr target :type (:object-instance (actor object-class))))
             :ui (:po ((External-Property :include (origin target)
                        :map-properties ((origin origin-obj) (target target-obj))))))))
 :actions ((:action CoachAssistance :mssg Mssg2Coach :parameters () :xform ((:confirm :implicit)
             :ui (:itec (:select ((MenuItem :name "Coach Assistance" :submenu Coach)))))
           (:action CoachHint ...)
           (:action identify-actor :mssg sar-create-actor :xform (:reusable (:visible always))
              :parameters ((:par noun-phrase :type (:object-instance noun-phrase)
                                 :ui (:itec ((ChoiceSelection :filter (:attributes name)
                                                              :prefix t :name "Noun Term"))))
                           (:par cardinality :type cardinality))
              :preconditions ((exist noun-phrase)))
           (:action identify-object-class ...)
           (:action identify-external-property :mssg sar-create-external-prop
              :parameters ((:par name :type :string)
                           (:par origin :type (:object-instance (actor object-class))
                              :ui (:itec ((MouseButton))))
                           (:par target :type (:object-instance (actor object-class))
                              :ui (:itec ((MouseButton :action Release)))))
              :preconditions ((exist (object-class actor) 2))
              :postconditions ()
              :xform (:reusable (:visible always)))))
```

Figure 3 – A high-level specification.

fied in terms of instances to be created at start up time. The data model concerns the application objects, their attributes and relationships. The control model specifies application actions, their parameters, pre- and postconditions, and relationships to other actions (task structure) and objects.

The specification provides four types of information: application semantics, initial context, UI hints, and design transformations. Only the first one is necessary, the other three are optional. Application semantics are defined in terms of the data and control models. The initial context can be defined only at the application level, while UI hints and design transformations can be defined at different levels, associated with the application component as a whole, or with the lower level components such as objects, actions and parameters.

The high-level specification is directly translated into an application model, and then into a UI, as is described in the following subsections. It is in this sense that the high-level specification serves as an external, persistent representation of the application and its UI that can be stored and communicated between the server and UI clients, and as such it facilitates interface transport. The specification fully defines the application model, but not the UI model. It contains only those UI details that we do not want MUSE to decide for us.

Because the main role of the specification is to serve as an external representation, we were concerned more with its content than with its form. It is not intended to be directly created and modified by UI designers, but through an interactive authoring tool, which is why we did not try to make it more user friendly.

Without going into all the details of the specification syntax, let us just say that each concept has a set of properties that can be defined, and each property is specified as a keyword/value pair "**keyword** value" where value can be a single token or a

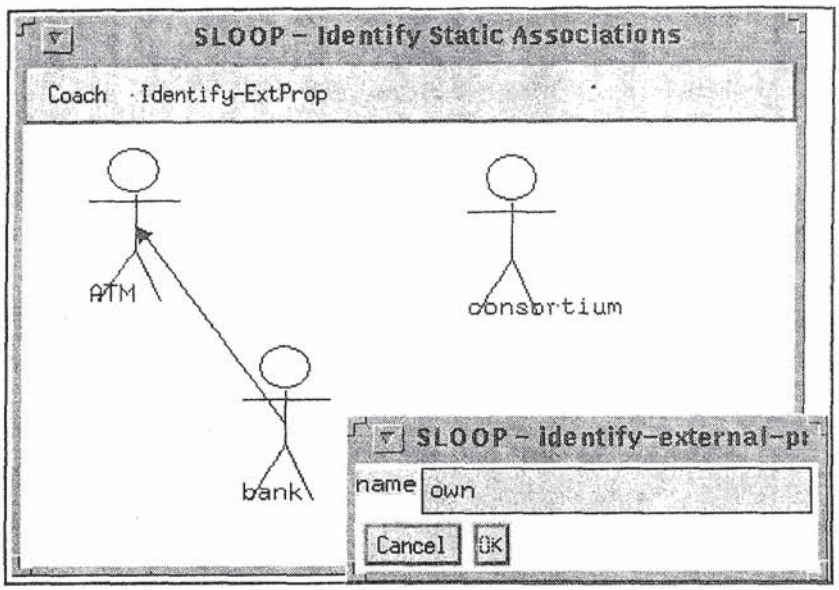

Figure 4 – UI generated for the specification in Figure 3

list of tokens. Keywords typically begin with ":" and are shown bold in the examples. For instance, the top level of specification corresponds to the application concept with keywords (properties) **:application-header** (for application name), **:ui** (UI hints), **:init** (initialization information), **:data** (user-defined data types), **:objects** (data model definition), and **:actions** (control model definition). The properties can be specified in any order.

The example shown in Figure 3 specifies that the application *StaticAssociations* has one user-defined data type (*cardinality*), data model comprising of 4 object classes (*noun-phrase*, *actor*, *object-class*, and *external-property*), and control model with 5 actions (*CoachAssistance*, *CoachHint*, *identify-actor*, *identify-object-class*, *identify-external-property*). Note that each object in turn also represents a user-defined type and can be used when defining object attributes and action parameters. For instance, the *actor* object has attribute *cardinality* of (user-defined data) type *cardinality*, as well as attribute *external-properties* of (user-defined object) type *external-property*. The object *external-property* also has attributes of object type, *origin* and *target*, which point to an instance of either *object-class* or *actor* object class. Each *external-property* must have one and only one value for attributes *origin* and *target*, which is the default for any attribute and parameter. On the other hand, each *object-class* or *actor* object class can have any number of relations, indicated by "**:set-max** nil" (no upper limit on number of values).

UI hints are defined at several levels in our example: at the application level they specify that the application will use for its main window a dialog box template *Graphical*, which will be named *Identify Static Associations* and will have a top level menu bar. UI hints for object classes specify what presentation objects to use and how to configure them. For actions, the hints specify what window to use, if needed, and for action parameters what interaction techniques to use and how to configure them. For instance, the *ChoiceSelection* technique allows specifying where to get items forming the list of choices and how to present the items. In the case of the parameter *noun-phrase* of the action *identify-actor*, the interaction technique knows (from the parameter type) that choices are instances of object class *noun-phrase* and the hint specifies that these instances be presented using the value of their attribute *name*. For the action *identify-external-property*, UI hints specify that both the *origin* and *target* parameters should be selected using a mouse (*MouseButton* interaction); to prevent ambiguities as to which attribute's value is selected, hints specify that a button release selects the target, and a button press (a default button operation) selects the origin.

Each practice activity in SLOOP can have a number of requests for different kinds of coach assistance and help. For simplicity, the example has only two coach actions, *CoachAssistance* and *CoachHint*, which will both be placed in a submenu *Coach* in the main menubar.

Application Conceptual Model

The UI is not generated directly from the specification, but from the application conceptual model (or aplication model, AM) instantiated based on the specification content. The AM provides internal representation of the information contained in the specification and is used by the generation, transformation, and consistency checking rules [Kovacevic 1992b].

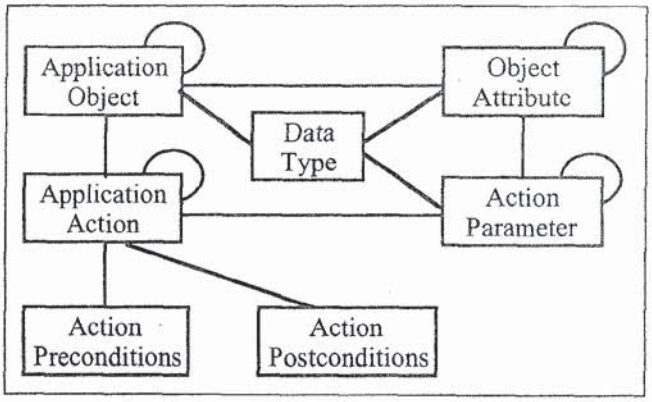

Figure 5 – Application conceptual model.

Figure 5 shows the seven entity types of the AM and their relations. Objects can be connected to other objects through inheritance and part/whole relations. Part/whole relations between actions can be used to define composite tasks. More details on the AM can be found in [Kovacevic 1992b]. Some of the relations shown are not necessary for the generation process, but provide information needed by the conceptual transformations [Foley 1987], which is why not all relations are defined in the specification in Figure 3.

Whereas the AM could be created and modified interactively, MUSE currently instantiates it solely from the specification. The initial context is provided by the coach component at run time, while the rest of the specification is defined at design time.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.