Vol. 44 No. 3 IPSJ Journal Mar. 2003

Regular Paper

Rule-Based Interactive Web Forms for Supporting End Users

Yoshinori Aoki,† Masahide Shinozaki†,☆ and Amane Nakajima†,☆

This paper describes techniques for developing assistance functions for Web form input operations. The techniques allow developers to define input-assistance functions as a set of assistance rules. A software module called a rule compiler converts these rules into a program that implements the assistance functions defined in them. The program is embedded into a Web form to monitor a user's input operations on a Web browser and to provide assistance functions such as help-message displays, validation checks of input values, and automatic inputs in accord with the user's input operations. By defining assistance functions as a set of assistance rules, developers can implement interactive Web forms without any complicated script programming. Therefore, it is possible to rapidly prototype assistance functions and reduce the development cost of the interactive Web forms. This paper explains the details of the assistance rules, and describes the design and implementation of a prototype system in detail. The prototype system includes a tool for defining assistance rules in a WYSIWYG (What You See Is What You Get) environment, and a form generation module including a rule compiler. The productivity of assistance function development and system performance are evaluated and discussed with reference to the prototype system. The results show that the prototype system can greatly reduce the cost of assistance function development, and that helpful forms can be generated very quickly.

1. Introduction

With the rapid growth of the Internet, electronic commerce has spread widely, allowing users to purchase many kinds of products and apply for services on the World Wide Web. Governments in many countries are actively pursuing electronic government initiatives, and it will be possible to submit application forms to government offices via the Web in the near future. When applying for such services on the Web, the user usually has to input the required information into a Web form, and then send it to a Web server.

Web-based services have brought users great convenience, since they can access the services at any time and from anywhere via the Internet. On the other hand, they often have to go through awkward steps to fill out complicated Web forms to apply for the services. As a result, some of them stop their input operations midway[26], or even change service providers to find more user-friendly services. USA Today wrote that 67% of Web transactions are abandoned at check-out, largely because top commerce sites have made few provisions for real-time, on-line customer service and support[19].

The following are the major reasons for this phenomenon:

(1) With the widespread use of personal computers, many novice users have become Internet users.

(2) Some Web forms are very complicated for end users. For example, there are too many input fields, the content is difficult because technical terms are used without explanations, or there are constraints between several input fields. Insurance policy forms or income tax forms are typical of these complicated forms.

(3) For many paper forms, there are other papers that show examples and explain matters that require special attention. However, such information is usually not integrated into Web forms because of the limitations of screen space and development costs.

To support end users on the Web, Web-based remote support systems have been proposed[3),23]. In those systems, an end user and a call center agent are connected via the Internet, and the call center agent supports the user by using real-time Web browser synchronization techniques. However, the operating costs of the call center become very expensive in such systems[45]. To solve the problem, it is very important to provide input-assistance functions that enable users to complete their inputs by themselves. Such services, which

† IBM Research, Tokyo Research Laboratory
☆ Presently with IBM Global Services - Japan

help users to solve problems by themselves on the Web, are called Web-based self-services, and have recently received considerable attention[14]. In our case, through provision of input-assistance functions, the number of users who connect to the call center could be reduced, and thus the operating costs of the call center could also be reduced, because fewer agents would be needed. In addition, users who could complete tasks by themselves with the help of the input-assistance functions will be satisfied, because they would not need to bother other people on the Web. However, the development would require a long time, making it expensive to provide such input-assistance functions, because there is no mechanism to easily implement such functions and complicated programming is needed. There is therefore a strong need to develop a method for implementing rich input-assistance functions in a short time at a low cost.

This paper describes a mechanism for developing an *assistance agent*, a software module for monitoring a user's input operations and for providing input-assistance functions. With this mechanism, developers can implement assistance agents without any programming by defining assistance rules. In this way, (1) development and maintenance costs will be reduced, (2) the assistance functions will satisfy end users by supporting their input, and (3) operating costs of the call center will be reduced, because the assistance agents will reduce the number of users who connect to the call center. We have developed a language for describing assistance rules, and a prototype system that automatically generates a program for an assistance agent including assistance functions defined in the rules. In the prototype system, a Web form in defined in three source files that define the (1) logical data structure, (2) form presentation, and (3) assistance rules. The prototype system generates a Web form from the three source files on the fly. The three source files are formatted in XML (Extensible Markup Language)[8]. The prototype system also includes a visual form design tool with which developers can define (1) XSLT (XSL Transformations)[9] stylesheets that define form presentations, and (2) assistance rules, both in a WYSIWYG environment. The tool also generates a rule file that automatically stores input values into the XML document. Hence, developers need not write a program for storing input values into the XML document. We have evaluated the prototype system from two aspects: (1) productivity of the assistance function development, and (2) performance of the form generation from the three source files.

The rest of the paper is organized as follows. The next section discusses related work. In the section after that, requirements are described. We then describe assistance rules and a mechanism for generating programs from assistance rules. The next section describes the design and implementation of the prototype system in detail. The last section presents our conclusions and plans for future work.

2. Related Work

On the average, 48% of the code of today's applications is devoted to the user interface[31]. Hence, many tools have been proposed to reduce the cost of user interface programming for interactive applications. For example, IntelligentPad[41] allows developers to build interactive applications by placing and connecting components called "Pads" on a screen. Peridot[30] is a PBD (Programming by Demonstration)[12),25] system with which developers can design a widget's presentation and its behavior by giving a demonstration. ITS[46] automatically generates interactive user interfaces by providing the dialog content and style rules. DEMO II[15] and Marquise[32] are PBD systems with which developers can create not only user interfaces, but also whole interactive applications by using demonstrations. These systems provide functions for defining presentations and behaviors of GUI (Graphical User Interface) components, and hence they are useful as general-purpose tools for building interactive user interfaces. However, we focus on developing input-assistance functions on the Web, and the functions provided by these general-purpose tools are not powerful enough to reduce the cost of assistance function development.

Mechanisms for help-system generation have been proposed to support the development of assistance functions. Mickey[43] automatically generates GUIs including help messages that guide users' GUI operations by means of help scripts written by developers, and hence conventional scripting work is needed. H3[29] generates default help systems from user interface specifications, and developers can improve the help systems by editing them. However, Web forms are usually written with an HTML au-

thoring tool and developers do not write specifications for each Web form. Sukaviriya et al. has been developing help systems (such as Cartoonist) to help users navigate by showing animated example operations[38),39)]. Showing example operations is useful and can also be done with our system. The Web Operation Recorder[2)] is useful for developing automatic presentations of sample operations from for the Web. Other functions such as input-value validation checks and automatic input are also needed to implement input-assistance functions for Web forms.

The SurfIt! browser[42)] provides interactive capabilities such as control of the visibility of input fields according to a user's input values. Such functions are implemented in Tcl/Tk script[35)], and the SurfIt! browser executes the script with its Tcl/Tk interpreter. Girgensohn proposed a Java-applet-based interactive Web form[16)]. In those systems, developers have to implement such interactive functions in conventional programming languages such as Tcl/Tk and Java. Consequently, the productivity of form development is very low.

Some technologies for reducing users' input operations for their personal information have also been proposed. P3P (The Platform for Privacy Preferences)[11)] is a standard that allows Web sites to express their privacy policies in machine-readable format. A P3P user agent, usually built into a Web browser, compares the privacy policy of a Web site with privacy preferences set by the user. If the user permits it, the P3P user agent can transfer the user's personal information to the Web site, so the user can skip over inputting that information. Microsoft Internet Explorer (IE) is a Web browser that remembers the past input values for Web forms. When a user tries to input data into a Web form again, the previous input value will be displayed. Internet Explorer for Macintosh Version 4.5 (IE for Mac 4.5) provides a personal information database where a user can register personal information such as a name, address, and phone number, and it automatically copies the information into Web forms. IE for Mac 4.5 parses and analyzes the HTML source of a Web form, and guesses which information is needed for each input field by using heuristics. For example, when the value of the name attribute of an input field is "first-name" or "FirstName," or the text "First name" appears near the input field, IE for Mac 4.5 regards the input field as requiring the user's first name and copies it from the personal information database. The Web site amazon.com (http://www.amazon.com) stores users' personal information such as names, addresses, and credit card numbers. When a user has previously bought something at amazon.com, he or she can check out without entering personal information (the 1-Click patent[21)]). These technologies are complementary to ours, because Web forms for personal information are not very complicated and these auto-input technologies are practical enough for such uses.

Microsoft FrontPage is a commercial HTML authoring tool, with which developers can add validation capabilities for input fields. For example, the range of a value can be set to be from 20 to 100. However, the validation is performed by using proprietary information written in comment tags in the HTML source file. Hence, the assistance function works only when the Web site was developed by using Microsoft products.

Several XML-based form languages have also been proposed. XFDL[7)] and XForm[24)] were created for defining Web forms in XML. XFDL and XForm are XML-compliant languages that include vocabulary for defining input fields. XForm provides a mechanism for generating Web forms from any XML document. However, an XFDL-aware or XForm-aware browser is needed to use them. XForms 1.0[13)] is an XML-based form description language being actively discussed by the W3C (World Wide Web Consortium). With such languages, the data structure and presentation of Web forms can be defined. These languages can be used with our technology, because our assistance rules are separated from the data structure and presentation of the Web forms.

3. Requirements

This section describes the requirements for designing and implementing Web forms with assistance functions.

- **Productivity:** Web forms are formatted in HTML[36)], and there are various commercial HTML authoring tools for developing simple Web forms. However, the costs of development and maintenance tend to be large, because a lot of programming is needed to implement assistance functions for each Web form. Web applications are generally required to be built in a very short time[10)]. Hence, methods for devel-

oping assistance functions in shorter times are strongly needed.

- **Defining assistance functions by business managers:** Developers have to be familiar with the details of all input fields on Web forms, such as constraints between input fields, ranges for input values, and information to be referred to for input fields. However, in many cases, the people who know the details are not developers, but business managers. Hence, it is necessary to provide functions for business managers to define assistance rules by themselves (or with developers). To provide such functions, a visual tool is needed with which business managers who have no programming skills can visually define assistance rules.
- **No special installation:** Assistance functions should work with normal Web browsers without any plug-ins. An original proprietary Web browser has been proposed to implement assistance functions in a Web browser[42]. Plug-in-based approaches have also been developed in some commercial form products such as Adobe Acrobat (http://www.adobe.com) and Accelio FormFlow99 (http://www.accelio.com). In such systems, the proprietary Web browser or the plug-in has to be installed in the client PC in advance. However, such installations bother end users, especially novice users. Assistance functions should be available without any installations whenever the need arises.
- **Easy integration with backend applications:** Input values in Web forms are usually processed by backend applications. Hence, it is necessary to provide a mechanism for integrating a Web server with backend applications.

4. Rule-Based Interactive Web Forms

This section explains mechanisms for developing assistance functions by defining assistance rules. First, it discusses approaches for implementing assistance agents, then describes the assistance rules, and finally explains a mechanism for generating an assistance agent from assistance rules.

4.1 Assistance Agent

An assistance agent monitors a user's input operations, and provides assistance functions. For example, the agent shows information such as a price table, advice or warning messages, and examples, checks the validity of input values, and automatically inputs specific values into specific input fields. The behavior of the assistance agents are defined by assistance rules. However, the functionality of the assistance agents partially depends on the implementation, so this section discusses alternative implementations before the design of the assistance rules.

4.1.1 Server-Side and Client-Side Assistance Agents

Assistance agents can be classified into two types according to where they work: (1) server-side assistance agents and (2) client-side assistance agents. Their features are as follows:

- **Server-side assistance agent:** This type of agent works as a server-side application. The agent validates input values and dynamically generates the next Web form including advice or warning messages after the previous Web form has been submitted by a Web browser. Hence, this type of assistance can be called "post-submission assistance."
- **Client-side assistance agent:** This type of agent works as a client-side application. The agent monitors a user's operations in a Web browser, and validates an input value when a user inputs it. The advice or warning messages are displayed depending on the current focus. This type of assistance can be called "pre-submission assistance," because the agent provides assistance functions when a user is entering data.

The server-side and client-side assistance agents are complementary, and neither of them can cover all assistance requirements by itself. When a backend database is needed to provide assistance functions, only the server-side assistance agents can offer such capabilities. For example, amazon.com (http://www.amazon.com) keeps each user's purchase history in their database, and allows users to purchase books and CDs without entering their personal information again by retrieving that personal information from the purchase history database. Another advantage of server-side assistance agents is that they can work persistently at a server even when the client PC is disconnected or turned off. Frameworks for developing such server-side agents have already been proposed[40),44)], and server-side assistance

agents are implemented in many Web sites.

On the other hand, no framework for developing client-side assistance agents has been proposed, and few Web sites provide client-side assistance agents, although the client-side assistance agents offer the following advantages:

(1) Client-side assistance agents monitor users' actions in a Web browser, and hence they can grasp the users' behavior in detail and catch the triggers for assistance.

(2) Server-side assistance agents provide assistance functions after form submission, and therefore usually give advice for all input fields included in the previous submission at one time. In other words, users cannot be assisted until they submit their forms. On the other hand, a client-side assistance agent provides assistance functions for each input field as a user is inputting data into the field. Hence, the user need not wait for the form submission before receiving assistance, and the total form-input time will be shortened.

This paper explains mechanisms for implementing client-side assistance agents. In the rest of this paper, an "assistance agent" means a client-side assistance agent.

4.1.2 Implementations of Assistance Agents

There are several approaches for implementing client-side assistance agents. This section compares approaches for determining what types of program should finally be generated from the assistance rules.

The following are the four major approaches for implementing assistance functions on Web forms, and the features of each:

- **Proprietary Web Browser:** In this approach, a proprietary Web browser is developed, and the browser has capabilities for interpreting and executing assistance rules. The SurfIt! browser[42] was developed in accordance with this approach, and it contains a Tcl/Tk interpreter and provides functions for dynamically inserting and deleting input fields with Tcl/Tk scripts.
- **Plug-in and Development Kit:** In this approach, end users have to install a special plug-in for their client PCs, and the plug-in displays forms on a Web browser. Adobe Acrobat and Accelio FormFlow99 are commercial products developed in accordance with this approach. They provide their own development kit for designing forms, and the forms are not formatted in HTML, but in a proprietary format.
- **Java Applet:** In this approach, forms are developed as Java applets and displayed by Web browsers. It is possible to implement complicated assistance functions in Java. Girgensohn et al. proposed Java-applet-based forms that can dynamically change the structure of the forms[16].
- **JavaScript:** In this approach, forms are formatted in HTML and the assistance functions are developed in JavaScript. This makes it possible to control the values, colors, sizes, and visibility of each input field by using Dynamic HTML functions[17], and the interfaces are standardized in the DOM (Document Object Model) specification[4].

Many of the Web browsers developed at the dawn of the Web era had only the capability of displaying HTML pages, and did not include application platforms such as Java runtimes and JavaScript interpreters. Therefore, developing a proprietary Web browser was the most popular approach until Web browsers come to include application platforms such as Java runtimes, JavaScript interpreters, and plug-in interfaces . Today, however, it has become very costly to develop proprietary browsers that fully support recent standards such as HTML, HTTP, XML, JavaScript, Java, and SSL. Recent Web browsers have many functionalities supported by Java runtimes, JavaScript interpreters, and plug-in interfaces with which we can develop assistance functions. Another disadvantage of the proprietary-browser and plug-in approaches is that end users have to install the proprietary browser or the plug-in on their client PCs in advance, while on the other hand de-facto standard Web browsers are installed on contemporary PCs. Therefore proprietary-browser and plug-in approaches are not currently suitable for developing assistance functions.

Java-applet-based forms have an advantage in that they work on normal Web browsers without any software or plug-in installations. However, a disadvantage of this approach is the difficulty of using Web servers to generate forms dynamically. In many Web sites, Web forms are dynamically generated according to the user's purchase history or input val-

ues for the previous form. Many technologies are currently available for generating dynamic Web pages on the server side, such as Java servlets[22], CGI (Common Gateway Interface)[20], and JSPs (Java Server Pages)[6]. However, when forms are developed as Java applets, such technologies are not suitable, because the technologies do not provide capabilities for generating Java-applet-based forms dynamically.

Recent versions of HTML and JavaScript allow us to control input fields; as a result, it has become possible to implement complicated assistance functions. Assistance functions implemented in JavaScript work on normal popular Web browsers without any software or plug-in installations, and they can be dynamically embedded into a Web form generated by server-side programs. Therefore, this approach is the most suitable of these four approaches to satisfy the requirements described in Section 3. The rest of this paper describes a mechanism for transforming assistance rules into a JavaScript program and embedding it into an HTML-based Web form.

4.2 Assistance Rules

This section describes the details of assistance rules. First, the interaction between a user and a Web form is modeled, and assistance rules are then designed according to the model.

4.2.1 Model

Figure 1 shows a typical cycle of user interface development. In our mechanism, prototyping is automatically done by generating an assistance agent from assistance rules. Hence, rapid definition becomes the key to rapid form development. In addition, as described in Section 3, the assistance rules must be simple enough for business managers who have no programming skills to define the rules.

Therefore, we adopted an event-action model, which is a very simple model in which an assistance rule can be defined as a pair of an event and a set of one or more actions. When a user performs an operation defined as an event in an assistance rule, the corresponding actions will be executed. Conditions are defined if the actions should be executed only when some conditions are satisfied. Hence, an assistance rule is described as a production rule[5] including an event, conditions, and actions. A production rule forms an if-then clause, in which the condition part is called the left-hand-side (LHS) and the execution part is called the right-hand-side (RHS). In our assistance rule, an LHS includes

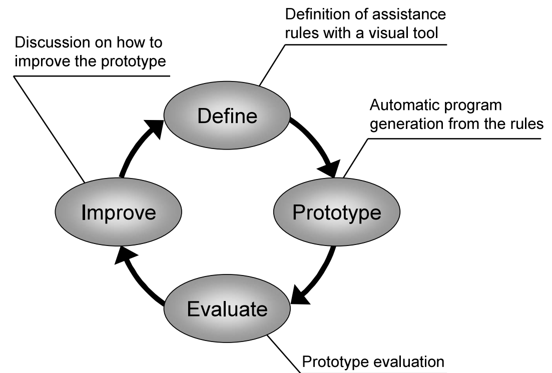


Fig. 1 Development cycle.

an event and conditions, and an RHS includes actions.

4.2.2 Events, Conditions, and Actions

When a user performs specific operations on an object in a Web form, the corresponding event will be issued by the object. In this context, an object can be an input field or a Web form itself, and an object firing an event is called an *event source*. In an assistance rule, an event can be expressed as a set of an event type, an event source, and parameters if needed. The details of each event and its expression are shown in **Table 1**.

In a condition part, conditions on input values can be described by using arithmetic and logical operations. Actions can be customized according to an input value by setting a condition on a input value. For example, the message "Please input expiration date in the format MM/YY" will be displayed when the value of input field "Payment" becomes "Credit Card," and the input field "Expiration date" will be disabled when the value of the input field "Payment" becomes "Money Order."

A target object for an action is called an *action target*. **Figure 2** shows three examples of input assistance functions with their action targets: a balloon help display, a Web page displayed for reference, and an enable/disable control. In an assistance rule, an action part can include a sequence of actions. For example, an assistance agent can first input a default value automatically, then disable the input field to prevent the user from updating it, and then show an alert message to let the user know the input field is already set. In an action part, each action can be expressed as a set of an action type, an action target, and parameters if needed. **Table 2** shows the details of each ac-

Table 1 Events.

Event	Description	Expression
Load	Page loading is completed.	<load/>
Reset	The form is reset.	<reset/>
Submit	The form is submitted.	<submit/>
Displayed	The source object is scrolled into the visible page area.	<displayed source="object1"/>
Disappeared	The source object is scrolled out of the visible page area.	<disappeared source="object1"/>
Mouse-over	The mouse pointer is over the source object.	<mouse-over source="object1"/>
Mouse-out	The mouse pointer leaves the source object.	<mouse-out source="object1"/>
Focus	The source object has the focus.	<focus source="object1"/>
Blur	The source object loses the focus.	<blur source="object1"/>
Change	The value of the source object is changed.	<change source="object1"/>
Input-string	A text string is input into the source object.	<input-string source="object1"/>
Input-number	A numeric value is input into the source object.	<input-number source="object1"/>
No-change	The source object loses the focus without a change of value.	<no-change source="object1"/>
Stop-input	Stop input operations to the source object for duration.	<stop-input source="object1" duration ="100"/>
Repeated-focus	Repeated focus on the source object.	<repeated-focus source="object1" times="3"/>
Repeated-input	Repeated input to the source object.	<repeated-input source="object1" times="3"/>
Specified-input-order	Input into objects in the specified order.	<specified-input-order> <order source="object1"/> <order source="object2"/> <order source="object3"/> </specified-input-order>
Specified-focus-order	Focus on the objects in the specified order.	<specified-focus-order> <order source="object1"/> <order source="object2"/> <order source="object3"/> </specified-focus-order>

Table 2 Actions.

Action	Description	Expression
Balloon help	Display the message in a balloon next to the target object.	<balloon-help-on target="object" balloonId="balloon1">message</balloon-help-on>
		<balloon-help-off balloonId="balloon1"/>
Alert window	Display the message in an alert window.	<alert-window>message</alert-window>
Web page	Display a Web page in a floating frame next to the target object.	<web-page-on target="object" pageId="page1" href="url"/>
		<web-page-off pageId="page1"/>
Auto-input	Input the value automatically into the target object.	<auto-input target="month">value</auto-input>
Click	The target object received a mouse click.	<click target="object"/>
Enable/Disable	Enable (disable) the target object to receive (refuse) a user's input.	<enable target="object"/>
		<disable target="object"/>
Visible/Invisible	Make the target object visible (invisible).	<visible target="object"/>
		<invisible target="object"/>
Highlight	Highlight (or remove highlighting from) the target object.	<highlight-on target="object" color= "color1"/>
		<highlight-off target= "object"/>
Scroll-in	Scroll the Web page to position the target object within the visible page area.	<scroll-in target="object"/>
JavaScript	Execute the JavaScript code.	<javascript>JavaScript code</javascript>

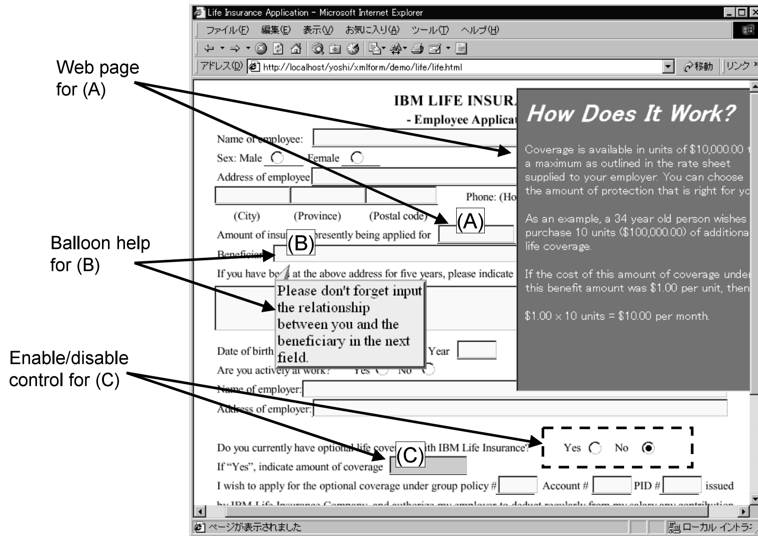
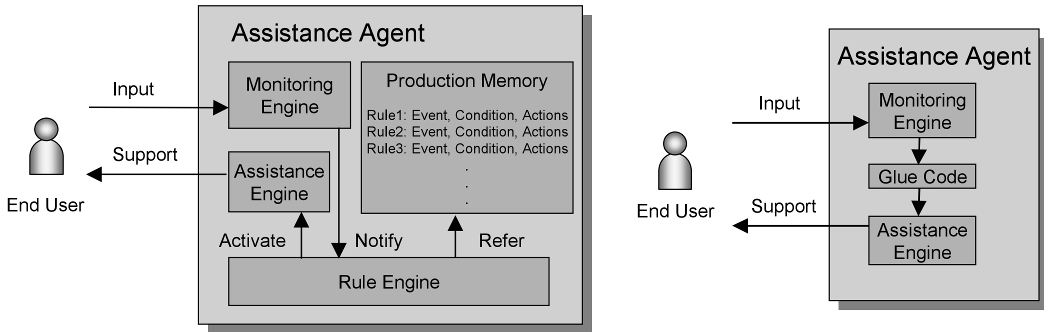


Fig. 2 Examples of assistance functions.



(a) Rule-engine-based Assistance Agent (b) Rule-compiler-based Assistance Agent

Fig. 3 Architectures of assistance agents.

tion and some sample expressions.

4.3 Assistance Agent Generation from Assistance Rules

This section explains a mechanism for generating an assistance agent from assistance rules. As explained in Section 4.2, an assistance agent is generated as a JavaScript program.

4.3.1 Architecture of Assistance Agents

There are two alternative architectures for implementing assistance agents: rule-engine-based and rule-compiler-based.

(1) **Rule Engine:** Assistance rules are interpreted at run time by an assistance agent in a Web browser.

(2) **Rule Compiler:** Assistance rules are interpreted and converted into a program at form-generation time by a Web server.

Figure 3 (a) shows the architecture of a rule-engine-based assistance agent. As shown in Fig. 3 (a), the assistance agent includes a monitoring engine that monitors a user's input operations on a Web form and an assistance engine that provides assistance functions for the user. In addition, the assistance agent also includes a production memory and a rule engine. The production memory holds the assistance rules, and the rule engine receives events from the monitoring engine, searches for related assistance rules in the production memory, and controls the assistance engine. In this architecture, the system need not dynamically generate an assistance agent, because the same assistance agent serves for all Web forms by receiving assistance rules for each Web form. In addition, the rule-engine-based assistance agents are extensible, because developers can add new functions by

extending the rule engine. For example, by extending the rule engine it is possible for the assistance agent to learn a user's habits by analyzing operation histories and dynamically add or modify assistance rules at run time. On the other hand, since the JavaScript programs are executed by an interpreter, a rule-engine-based assistance agent may cause a performance problem, especially in searching for assistance rules. Another disadvantage is the program size. A rule-engine-based assistance agent is extensible, but when a rule engine is too complicated, the assistance agent program becomes large, which results in a long download time. agent.

Figure 3 (b) shows the architecture of a rule-compiler-based assistance agent. The agent includes glue code that binds each action to an appropriate event and conditions. In the rule-compiler-based assistance agent, relationships between events, conditions, and actions are hard-coded into the glue code, and therefore the extensibility of rule-compiler-based assistance agents is inferior to that of rule-engine-based assistance agents. On the other hand, rule-compiler-based assistance agents are superior to the rule-engine-based ones in regard to system performance and compact program size. Since we attached greater importance to system performance than to extensibility, we have implemented a rule-compiler-based assistance agent in our prototype system.

4.3.2 Rule Compiler

This section explains the mechanism of our rule compiler, which automatically generates an assistance agent as a JavaScript program from assistance rules.

The rule compiler receives a set of assistance rules as its input, and generates a JavaScript program that implements the assistance functions defined in the assistance rules as its output. The behaviors of the rule compiler are as follows:

(1) **Syntax Check:** Reads assistance rules and checks their syntax before generating programs.
(2) **Conflict Check:** After reading an assistance rule, the rule compiler checks whether there are other assistance rules that include exactly the same event and conditions. If the rule compiler finds such rules, it shows a warning message for the developers to confirm whether the rules should be executed sequentially or whether they are merely conflicting.
(3) **Object Check:** This check is optional. When the rule compiler receives an HTML document into which an assistance agent is embedded as its input, the rule compiler confirms whether the HTML document includes all the event sources and action targets. This check is useful when Web forms are dynamically generated, because if a generated Web form does not include event sources or action targets, the assistance agent will show error messages to the end users. Since Web servers cannot detect and avoid such client-side errors, the users will be confused rather than helped by seeing them.
(4) **Assistance-agent Generation:** After passing checks (2) and (3) above, the rule compiler generates an assistance agent as a JavaScript program.

The rule compiler performs the above steps by constructing a *DRR-Tree (Dependency Relationship Resolution Tree)* as shown in **Fig. 4**.

The DRR-Tree contains a tree structure that defines the relationships between input-field objects and the flow of the assistance functions defined in the assistance rules. The rule compiler generates the input-field objects only when the rule compiler receives an HTML document as its input, and the input-field objects are generated in accord with the DOM[4]. After the rule compiler reads one assistance rule, it generates a subtree that expresses the flow of an assistance function including an event, conditions, and actions. The rule compiler attaches the subtree to an input-field object in the DRR-Tree. In Fig. 4, the subtrees are shown as event handlers, and the colored subtree represents one assistance rule and means that when a numeric value is input into input field A (the corresponding input-number event is fired) and the input value is smaller than 20 (condition), a value is automatically input into another input field and the input field disabled so that it cannot be modified by the user (actions). In Fig. 4 the actions for input values of more than 20 are omitted, and the conditions and actions for the input-string event are also omitted. As Fig. 4 shows, an assistance agent detects the events defined in Section 4.3.2 by hooking into HTML events[17),34)]. For example, when a user changes an input value on a Web form, an HTML event, change, will be fired. An assistance agent hooks the change event, and if the

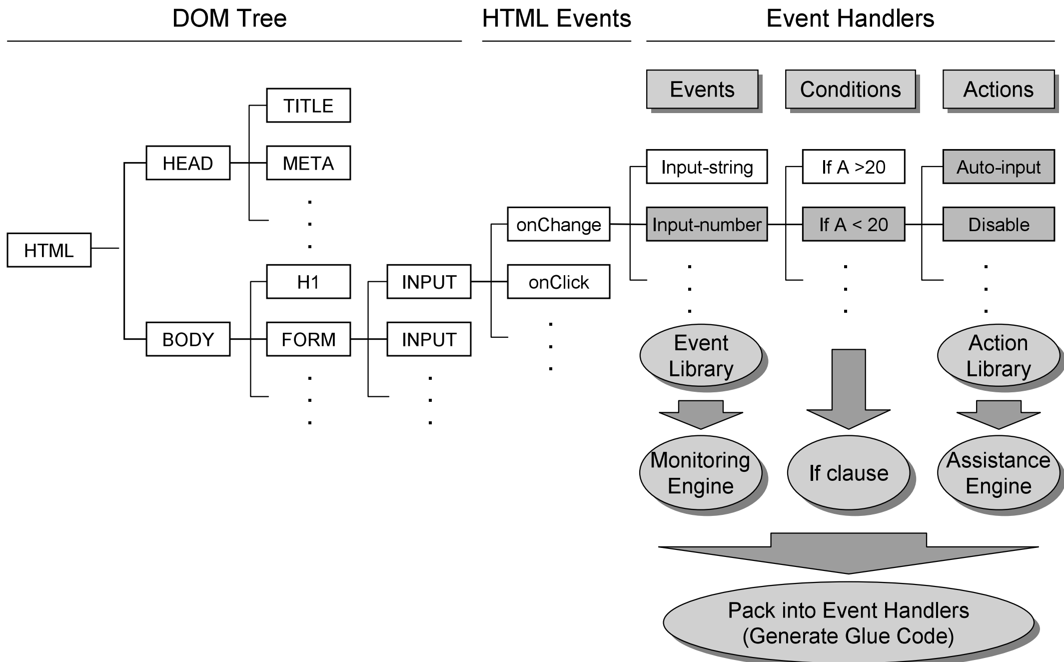


Fig. 4 Dependency relationship resolution tree generated by rule compiler.

new value is a text string, the assistance agent fires an input-string event. If the value is a numeric value, the assistance agent fires an input-number event in the same way. These event detections are executed by calling an event-detection method in the change event handler. The rule compiler contains a set of such event-detection methods as an event library, and each event-detection method is statically bound with an HTML event. When the rule compiler finishes reading all of the assistance rules, a complete DRR-Tree has been constructed. In the process of the construction of the DRR-Tree, the conflict and object checks are completed.

When a complete DRR-Tree has been constructed, the rule compiler generates a JavaScript program. To generate the program, the rule compiler gathers the required event-detection methods from the event library and assembles them as a monitoring engine. The rule compiler also contains an action library that includes methods for all kinds of actions, and it gathers all action methods used in the DRR-Tree and assembles them as an assistance engine. After transforming conditions into JavaScript if clauses, the rule compiler finally packs all related event-detection-method calls, if clauses, action-method calls, and their parameters into an event handler and adds it to the event handler for the input-field. These packing operations are executed for each HTML event of each input field, and the generated code is the glue code of the assistance agent.

5. Implementation

This section explains the design and implementation of our prototype system.

5.1 Form Definition in XML

In the prototype system, Web forms are defined in XML. This section describes problems with current form definitions in HTML, and explains how to define forms in XML.

5.1.1 Problems with Form Definitions in HTML

HTML is a language for describing Web pages, and it also allows us to put input fields on the Web page[36]. Another option embeds script programs by using the <SCRIPT> tag. When we define a Web form in HTML, the data structure of the form, the presentation, and programs are all mixed in the HTML file. Therefore the HTML file tends to have high dependency among data structures, presentation, and programs. This increases the cost of form development and maintenance.

5.1.2 Form Definition in XML and Its Features

In the prototype system, a Web form is defined with three separate XML files, which separately define data structures, the presentation,

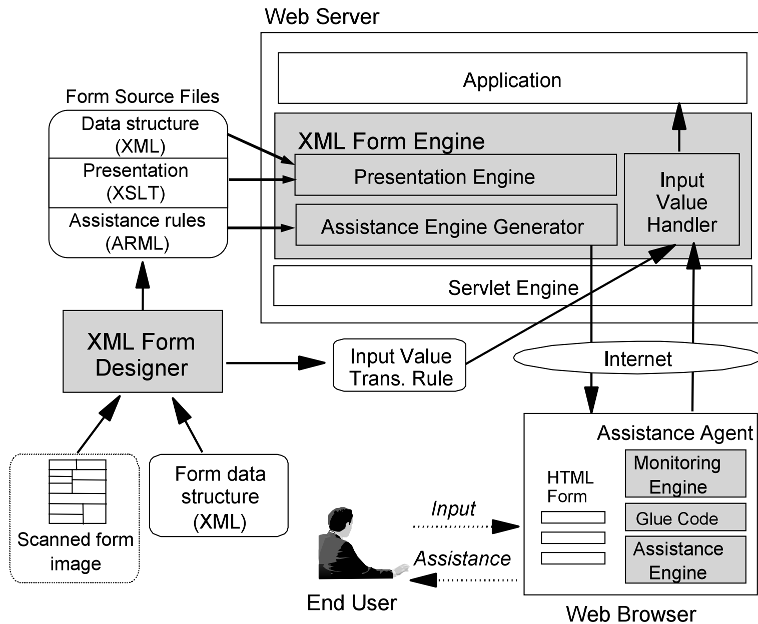


Fig. 5 System overview.

and assistance rules. The concrete, logical data structure of the form is defined as an XML document, and the form presentation is defined as an XSLT stylesheet that transforms the XML document into an HTML form. Assistance rules are defined as an ARML rule file. ARML (Assistance Rule Markup Language)[1] is an XML-compliant language that we have developed to describe assistance rules for Web form input. By defining ARML rule files, we can develop Web forms with rich input assistance capabilities more quickly than in conventional development with script programming.

The following are the major advantages of form definition in XML:

- **Connectivity:** XML does not depend on any operating system or programming language, and is supported by many of the recent commercial middleware systems. It is therefore very suitable for exchanging form data with backend applications, including databases.
- **Syntax Check:** By defining a DTD (Document Type Definition)[8] for ARML, an XML parser can validate the syntax of ARML rule files. Therefore a proprietary syntax checker need not be developed.
- **Multiple form Layout Development:** When multiple Web forms have to be provided for various devices such as PCs, PDAs, and cellular phones, an XML document that defines the data structures and any application that handles the XML document can be shared. By defining multiple XSLT stylesheets, form presentations and formats can be flexibly switched. By defining multiple ARML rule files, assistance functions can potentially be generated according to the device. However, the current prototype system supports only Web browsers for PCs.

5.2 System Overview

Figure 5 shows an overview of the prototype system. The system provides the colored components. The two main ones are:

- XML Form Designer
- XML Form Engine

With the XML Form Designer, developers can visually create an XSLT stylesheet, an ARML rule file, and an input-value-transformation rule file. The XSLT stylesheet transforms an XML document into an HTML form, and the input-value-transformation rules store input values in the XML document. The ARML rule file defines assistance rules for the users' Web form input.

The XML Form Engine is a class library that provides common functions to process Web forms with XML documents, and provides the following functions:

(1) Form generation from XML, XSLT, and ARML files.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE assistance-rule-set SYSTEM "arml.dtd">
<assistance-rule-set>
  <assistance>
    <event>
      <mouse-over source="ap-name"/>
    </event>
    <condition>
      <eq>
        <value-of element="ap-name"/>
        <value/>
      </eq>
    </condition>
    <action>
      <balloon-help-on target="ap-name" balloonId="b01">Please input your
      name in the following format, "Family, Given."</balloon-help-on>
    </action>
  </assistance>
  <assistance>
    <event>
      <mouse-out source="ap-name"/>
    </event>
    <action>
      <balloon-help-off balloonId="b01"/>
    </action>
  </assistance>
</assistance-rule-set>
```

Fig. 6 Example ARML rule file.

(2) Storing input values in an XML document.

A Web form, generated in the XML Form Engine, is formatted in HTML, and includes JavaScript programs that implement the input assistance capabilities. The JavaScript programs work in a Web browser, and detect a user's input operations and then provide input assistance for the user. The main class of the XML Form Engine is implemented as a Java class that extends the servlet class[22].

Application developers can efficiently create complicated Web forms by using the XML Form Designer and the XML Form Engine. The following subsections explain the details of ARML, the XML Form Designer, and the XML Form Engine.

5.3 Assistance Rule Markup Language

ARML is a language developed for defining assistance rules for Web form input. The syntax of ARML is based on XML. In XML, when a DTD is defined and declared at the top of an XML document, an XML parser can validate the XML document to check whether the XML document conforms to the DTD[27]. Therefore, by defining the DTD for ARML, an XML parser can check the syntax of any ARML rule files.

Figure 6 shows a very simple example of an ARML rule file. The ARML rule file includes only two assistance rules, which are defined by using the <assistance> tags. The first rule in Figure 6 says that the text string, "Please input your name ...," will appear next to the input field "ap-name" in a balloon when the mouse pointer is over the input field "ap-name" and the value of "ap-name" is null. The second rule says that the balloon will disappear when the mouse pointer moves away from the input field "ap-name."

5.4 XML Form Designer

The XML Form Designer is a standalone application with which developers can create

(1) an XSLT stylesheet for transforming an XML document into an HTML form,
(2) input-value-transformation rules for storing input values in the XML document, and
(3) an ARML rule file for defining assistance rules.

5.4.1 Presentation Definition

By using the XML Form Designer, developers can visually bind XML elements or attributes to input fields, such as text fields, radio buttons, or combo boxes, in a WYSIWYG (What You See Is What You Get) environment. They can define the presentation of the Web form by changing the attributes of the bound input fields, such as position, size, and colors. In addition to that, the XML Form Designer allows us to use a scanned image of a paper form as a background image for a Web form. By locating the input fields on the image, developers can build a Web form that provides a paper-form-like view, as shown in **Figure 7**. In this case, the XML Form Designer generates CSS (Cascading Style Sheet)[18],[28] attributes for each input field to define position, size, and so on.

After the form presentation has been defined, the XML Form Designer generates the following two XSLT files:

(1) An XSLT stylesheet that transforms the XML document into an HTML form. This file also includes the CSS format attributes for the input fields, such as position, size, and color.
(2) An XSLT file of transformation rules for formatting and storing input values in the XML document.

Figure 8 shows a screen shot of our implementation of the XML Form Designer as implemented with JDK 1.2.2. In the top right

(a) Paper Form

(b) Paper-like Web Form

Fig. 7 Paper-like form view.

Fig. 8 Screen shot of the XML form designer.

area, called the *Tree Viewer*, we can see an XML document with its tree structure. In the left area, called the *Layout Panel*, we can position the input fields. In the bottom right area, called the *Presentation Editor*, we can change the attributes such as location, color, and font. The location and size attributes can also be changed directly by using the mouse in the Layout Panel.

The steps in creating a form are as follows: First, an XML document has to be loaded into the Tree Viewer. Next, we have to select an XML element or attribute from the Tree Viewer, and we can create an input field on the Layout Panel. We can change the input field type and the attributes for each input field. After that, we generate the two XSLT files as described above, the XSLT stylesheet and the input value transformation rules. In the XML Form Designer, we select an XML element or attribute, and then create an input field. In this way, the XML Form Designer establishes the relationships between XML elements and input fields, and the XML Form Designer generates XSLT files from these relationships.

Figure 9 shows a very simple example of an XML document, and **Fig. 10** and **Fig. 11** show examples of XSL templates included in a corresponding XSLT stylesheet generated from the XML Form Designer. In this example, the user

```
<Person>
  <Name></Name>
  <Gender></Gender>
</Person>
```

Fig. 9 Example of an XML document.

```
<xsl:template match="/">
  <HTML>
  <HEAD>
  <TITLE>Insurance Policy Form</TITLE>
  </HEAD>
  <BODY BGCOLOR="#FFFFFF">
  <IMG ID="FormImage" SRC="FormImage.gif"
   STYLE="position:absolute; left:0pt; top:0pt;"/>
  <FORM NAME="app" METHOD="POST" ACTION="/FormServlet">
    <xsl:apply-templates select="/Person[1]/Name[1]"/>
    <xsl:apply-templates select="/Person[1]/Gender[1]"/>
  </FORM>
  </BODY>
  </HTML>
</xsl:template>
```

Fig. 10 Example of an XSL template for transforming a tree structure.

designed a form that includes one text field and two radio buttons. The stylesheet transforms the “Name” element into a text field, and the “Gender” element into “Male” and “Female” radio buttons. Though the focus of this paper is on the steps involved in handling assistance rules, the XML Form Designer actually plays an important role in linking the abstract labels of the XML to the concrete values the XML documents will receive. In this example, XML Form Designer is used to specify that the “Gender” element will have two options named “Male” and “Female”, and that the “Name” element will be a text field.

To generate the XSLT stylesheet, the XML Form Designer first generates an XSL template that transforms the tree structure of the XML document shown in Fig. 9 into the tree structure of an HTML document, as shown in Fig. 10. The XML Form Designer then generates XSL templates that transform XML elements (or attributes) into input fields, as shown in Fig. 11. The templates include not only input-field types but also default values and CSS attributes such as positions, sizes, and colors. The default value of an input field depends on the value of the related XML element (or attribute). For example, the second template shown in Fig. 11 determines which of the “Male” and “Female” radio buttons should initially be checked, according to the value of the “Gender” element of the XML document in Fig. 9. In this example, neither button is initially checked, because the “Gender” element includes no value.

```
<xsl:template match="/Person[1]/Name[1]">
  <INPUT NAME="ap-name" STYLE="position:absolute; left:183; top:64;
   width:374; height:25; font-size:12; color:#000000; background:#ffffcc;
   visibility:visible; " >
   <xsl:attribute name="TYPE">
    <xsl:text>text</xsl:text>
   </xsl:attribute>
   <xsl:attribute name="VALUE">
    <xsl:apply-templates/>
   </xsl:attribute>
  </INPUT>
</xsl:template>

<xsl:template match="/Person[1]/Gender[1]">
  <INPUT NAME="ap-gender" STYLE="position:absolute; left:128; top:91;
   width:33; height:23; font-size:12; color:#000000; background:#ffffcc;
   visibility:visible; " >
   <xsl:attribute name="TYPE">
    <xsl:text>radio</xsl:text>
   </xsl:attribute>
   <xsl:attribute name="VALUE">
    <xsl:text>Male</xsl:text>
   </xsl:attribute>
   <xsl:if test="text()='Male'">
    <xsl:attribute name="CHECKED"/>
   </xsl:if>
  </INPUT>
  <INPUT NAME="ap-gender" STYLE="position:absolute; left:232; top:92;
   width:33; height:22; font-size:12; color:#000000; background:#ffffcc;
   visibility:visible; " >
   <xsl:attribute name="TYPE">
    <xsl:text>radio</xsl:text>
   </xsl:attribute>
   <xsl:attribute name="VALUE">
    <xsl:text>Female</xsl:text>
   </xsl:attribute>
   <xsl:if test="text()='Female'">
    <xsl:attribute name="CHECKED"/>
   </xsl:if>
  </INPUT>
</xsl:template>
```

Fig. 11 Example of an XSL template for generating input fields.

It is not easy for developers to create an XSLT stylesheet with a general text editor or XSLT editor, because the syntax of XSLT is complicated, as is shown by Fig. 10 and Fig. 11. By using the XML Form Designer, a developer can create both XSLT files, an XSLT stylesheet and the input value transformation rules, even if he or she is not familiar with XSLT syntax. In addition, the developer does not have to write any code for storing input values in the XML document, because the XML Form Designer automatically generates input value transformation rules. In these ways, this tool greatly reduces the cost of form-based application development.

5.4.2 Assistance Rule Definition

Figure 12 shows the ARML Editor implemented in the XML Form Designer. Developers can switch from the Presentation Editor in Figure 8 to the ARML Editor by selecting a tab. The ARML Editor consists of two parts: (1) *Assistance Rule Editor* and (2) *Assistance Rule List*. With the Assistance Rule Editor, a user can define a new assistance rule, which corresponds to adding a new <assistance> element

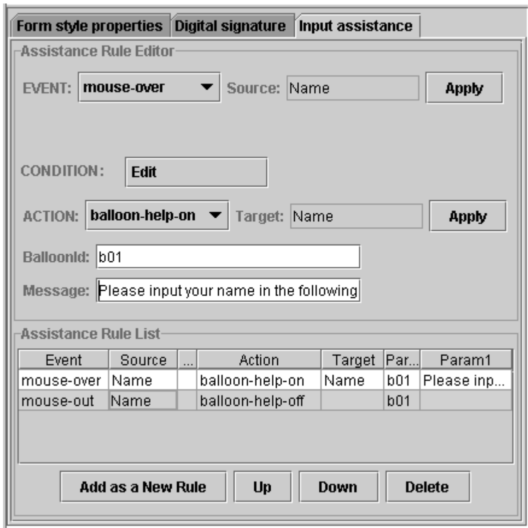


Fig. 12 ARML editor in the XML form designer.

in an ARML rule file. The user defines a new assistance rule in accordance with:

(1) Select an appropriate event type and its source object from the Tree Viewer or the Layout Panel, using direct manipulation[37], and input any required parameters.
(2) Edit the condition if necessary.
(3) Select an appropriate action type and its target object from the Tree Viewer or the Layout Panel using direct manipulation, and input any required parameters.

Using the Assistance Rule List, the developer can change the priorities of the rules, delete rules, or modify parameters. On the Assistance Rule List, each assistance rule is displayed on one line, and an assistance rule with higher priority is displayed higher in the list. When several actions are bound to one event, all of the actions are executed in order of their priority.

Since the ARML Editor generates the rules in the ARML format, developers need not become familiar with the detailed syntax of ARML to define assistance rules.

5.5 XML Form Engine

This section describes the details of the functions of the XML Form Engine.

5.5.1 Web Form Generation

When a Web browser requests a form, the XML Form Engine generates a Web form from the XML, XSLT, and ARML files. The form is generated in accordance with the following steps:

(1) Load the XML document and XSLT stylesheet, and generate an HTML-formatted Web form from these files.
(2) Load the ARML rule file and generate a JavaScript-based assistance agent that implements the assistance functions defined in the ARM rule file.
(3) Embed the assistance agent generated in Step (2) into the Web form generated in Step (1).

The XML Form Engine includes a *Presentation Engine* that performs Step (1), and an *Assistance Engine Generator* that performs Steps (2) and (3), as shown in Fig. 5.

The Presentation Engine includes a software module called an *XSLT processor* that generates an output XML document (in this case, an HTML form) from an input XML document and an input XSLT stylesheet. The Assistance Engine Generator includes a software module called the *ARML compiler* that generates script programs from an ARML rule file. The ARML compiler parses an ARML rule file and generates a Monitoring Engine that includes functions for detecting the events used in the ARML rule file. It also generates an Assistance Engine that implements the actions used in the ARML rule file. In addition to those engines, it also generates glue code that binds each action to an appropriate event. The Assistance Engine Generator generates the above programs in JavaScript by using the techniques described in Section 4.3.2.

5.5.2 Input Value Handling

When a user submits a Web form, the Web browser sends the pairs of input field names and values as a text string. An input field name is defined in an HTML file as a NAME attribute of an input field tag, such as an <INPUT> or <SELECT> tag[36]. The Web application needs to know all input field names and the locations in the XML document where the input values should be stored. Such information is usually given to a server-side program, such as CGI scripts or servlets, as program logic. Therefore, developers have to write many lines of code to store the input values in an XML document. Even a minor change in the form may require substantial programming effort to deal with the new or changed information.

We solve this problem by separating such information from the application logic. In the prototype system, such information is given to an application as an input value transformation rule file, as shown in Fig. 5. When a Web form is

submitted from a Web browser, the XML Form Engine refers to the input value transformation rules and automatically stores appropriate input values into an XML document according to the rules. The application then processes the XML document that already includes the input values. This automatic transformation is done in a software module called the *Input Value Handler* which is a part of the XML Form Engine. In the prototype system, input value transformation rules are also written in XSLT format.

5.5.3 Processing Flow of the XML Form Engine

Figure 13 shows an example of the flow of an application developed on top of the XML Form Engine. The application generates two Web forms from an XML document to gather data from a user. In Fig. 13, XML-0 represents an XML document that contains only initial values. When a user requests the first form, the XML Form Engine generates the first form (HTML-0) from XML-0, XSL-0, and ARML-0. When the user submits the first form, the XML Form Engine stores input values in XML-0 according to an input value transformation rule (TransRule-0). The application gets the XML document (XML-1) including input values from the first form from the XML Form Engine, and processes it as required. After that, the XML Form Engine generates the second form from XML-1, XSL-1, and ARML-1. In this way, all the application has to do is indicate which XML, XSL, and ARML files are to be used, and then process the returned XML document that includes the input values.

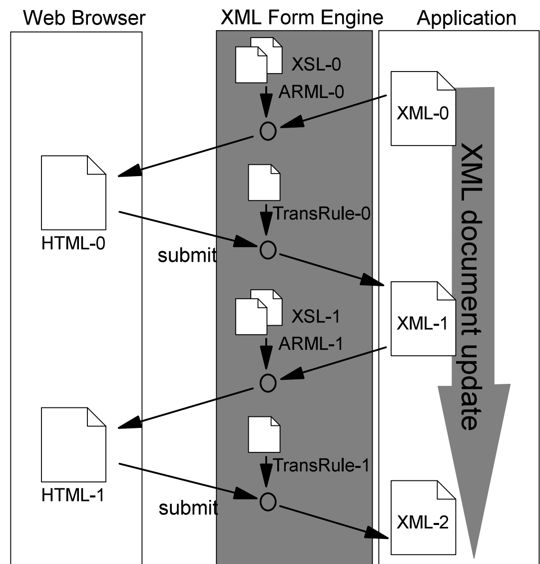


Fig. 13 Processing flow of the XML form engine.

6. Evaluation

By using the prototype system, this rule-based method for developing assistance functions for Web form input has been evaluated from two aspects: (1) productivity of the assistance function development, and (2) performance of the prototype system.

6.1 Productivity

This section describes qualitative evaluations of productivity by comparing our system and existing systems.

First, we compare our system and existing rule-based interface builders. PBD systems, such as Peridot[30], Marquise[32], and DEMO II[15], are typical rule-based systems for building interactive user interfaces. In these systems, developers define the behaviors of user interfaces by demonstrating example behaviors. The systems contain rules about layout constraints and domain specific knowledge. When developers demonstrate using the systems, the systems infer the developers' intentions to generalize the demonstrations and automatically generate programs. However, the systems often make mistakes in the inferences. Developers have to correct the mistakes manually, and they are usually time-consuming and sometimes impossible to correct[33]. On the other hand, our system focuses on a specific domain, developing input-assistance functions for Web forms, and it is possible for our system to provide vocabularies of events and actions that are frequently used to develop assistance functions for Web forms. Consequently developers are able to define the behaviors of assistance agents as sets of assistance rules, and our system never performs complicated inferences. Therefore it does not require modifications for incorrect inferences, and developers do not have to demonstrate many times to define one behavior. Mickey's approach[43] is similar to ours. However, developers have to write actual help scripts in Mickey. To reduce the cost of defining assistance rules, our system provides a visual tool, XML Form Designer, and the model of the assistance rules is very simple. In addition, our system is extensible in terms of vocabulary for events and actions. When new terms for events and actions are needed, it is easy to add them into our system. Such enrichment of the vocab-

ulary will enhance the value of our system.

We now present a rough comparison of the productivity of our method and a conventional Web programming style. It is not easy to precisely evaluate the productivity of assistance function development, because the productivity heavily depends on the developers' programming skills.

A total of 26 assistance rules are included in the source ARML rule file for the Web form shown in Fig. 2, and approximately 1,000 lines of JavaScript code, implementing the assistance functions, are embedded in the HTML document for the Web form. The program size for the assistance engine depends on what types of action are used in the rule set, and the program size for the monitoring engine depends on what types of event are used in the rule set. The amount of glue code is proportional to the number of assistance rules in the rule set, and only a few lines of glue code are needed for one assistance rule. In our experience, the ARML compiler generally outputs from several hundred to fifteen hundred lines of JavaScript code. A program of such a size generally takes a few weeks to develop in conventional script programming style, though, as just mentioned, it heavily depends on the programmer's skill. Even when a developer uses a commercial product for Web page design, such as Microsoft FrontPage or IBM Homepage Builder, the development cost of assistance functions will not be greatly reduced. This is because such products provide environments for script programming, but the environment cannot greatly reduce the amount of actual programming.

By defining assistance rules visually with the XML Form Designer, the assistance functions are automatically generated from the rules without any programming. In our experience, one assistance rule can be defined in several minutes. Therefore, only a few hours are needed to define the assistance rule set for one Web form. In fact, only 90 minutes were needed to develop the Web form shown in Fig. 2.

XML Form Designer is designed to work collaboratively with XML Form Engine. When developers use XML Form Engine, Web-form generation is performed automatically by using the XSLT stylesheets and ARML rule files generated from the XML Form Designer. Hence the developers do not need to write many lines of codes to generate the Web forms. In addition, XML Form Engine automatically stores input values into XML documents and sends them to applications by giving input-value-transformation rules generated by XML Form Designer. Therefore, developers do not have to write many lines of code to store input values in XML documents. For example, the Web form shown in Fig. 2 contains 41 input fields, and approximately 1,000 lines of code are needed if our tool is not used (many lines of code are usually needed for DOM tree traversal in XML programming).

Therefore the prototype system allows developers to greatly reduce the cost of assistance function development.

6.2 Performance

We evaluated the performance of both a generated assistance agent on a client PC and server-side Web form generation.

For the evaluation we used five Web forms, including the Web form in Fig. 2, as generated by the prototype system on a PC with a Pentium II 450 MHz CPU. In the results, we could not detect the overhead of the assistance agents working in the Web browsers. This is because the rule compiler in our system generates assistance agents as JavaScript programs that implement behaviors defined by assistance rules. Therefore, the behavior of the assistance agents are is hard-coded as JavaScript, and run-time rule evaluations are not performed on the client PCs.

Web forms can be generated from the XML, XSLT, and ARML files at design time when the Web forms are static. However, Web applications may have to generate a Web form on the fly at run time when the Web form depends on users' choices or on a previous form's input values. If Web forms are generated on the fly and the generation times are long, then users will have to wait for a long time before their Web browsers can display the forms. For this reason, we were concerned to evaluate the performance of our implementation of the XML Form Engine.

A form's generation time depends on the contents of the three source files: the XML document, the XSLT stylesheet, and the ARML rule file. For initial testing, we used the Web form shown in Fig. 2. The source XML document for this form includes 41 elements, and the source XSLT stylesheet includes 32 template rules. The Web form generated from those files includes 34 input fields. The source ARML rule file includes 26 assistance rules, and gen-

Table 3 System configuration.

HW and SW	Product Name, Spec.
CPU	Intel Pentium II 450 MHz
Memory	384 MB
OS	Windows 2000 Professional
Web Server	IBM HTTP Server Ver.1.3.6 (Customized Apache 1.3)
Servlet Engine	WebSphere Application Server Ver.2.03.1
Java runtime	IBM JDK 1.1.7

Table 4 Form generation time.

Item	Time (msec.)
XSLT Processor (Lotus XSL Ver.0.18.2)	70.4
ARML Compiler	40.5
Other	57.9
Total	168.8

erated approximately 1,000 lines of JavaScript code. We generated the Web form 10 times on the system described in **Table 3**, and **Table 4** shows the average times for form generation. In this evaluation, only one client was accessing the Web server.

Table 4 shows that the average form-generation time is 168.8 milliseconds, including 70.4 milliseconds for HTML generation in the XSLT processor and 40.5 milliseconds for JavaScript generation in the ARML compiler. Therefore, the performance of our prototype XML Form Engine is fast enough to generate Web forms on the fly.

7. Conclusions and Future Work

This paper has explained mechanisms for developing Web forms with input assistance functions. The assistance functions are defined as a set of assistance rules, and each assistance rule includes an event, conditions, and actions. The assistance rules are transformed into a JavaScript program by a software module called a rule compiler, and the JavaScript program is embedded into a Web form. The program monitors a user's operations on Web forms, and provides assistance functions according to events and conditions defined in the assistance rules.

This paper also summarizes our prototype system including a visual tool for defining assistance rules and the presentation of a Web form and a server-side software module for generating a JavaScript program from the assistance rules, producing an HTML-formatted Web form from an XML document and an XSLT stylesheet, and embedding the program into the Web form.

The evaluation of the prototype system shows that the productivity of Web form development using the prototype system is much higher than that of conventional Web form development using script programming. The system performance of the prototype system was also evaluated, and the results show that the prototype system can generate Web forms fast enough for practical use on a Web server.

There are two major future directions. One is extension of the vocabulary used in the current assistance rules. An assistance rule is based on an event-action model, and words for the events and actions are already defined. However, we have not evaluated the vocabulary by using generated Web forms with real users. When the system is evaluated with actual users, some of the current events and actions will be removed or updated, and the vocabulary of assistance rules will become more sophisticated. In addition, by collecting logs of many users' behavior, statistical analysis can be performed to find out how users behave when they encounter problems. The other direction of future work is abstraction of the assistance rules to support multiple devices such as PDAs and cellular phones. Some vocabulary items of the current assistance rule are designed on the assumption that end users use JavaScript-enabled Web browsers on PCs. However, devices other than PCs, such as PDAs and cellular phones should also be supported. To support such devices, abstractions of the assistance rules are needed to remove language and device dependencies. In addition, multiple rule compilers will also be needed. For example, a rule compiler for Web browsers on PCs generates a JavaScript program, but another rule compiler for Web browsers on cellular phones will generate a server-side program to provide the same assistance. These components will ultimately be provided as a framework for developing user supportive Web sites for multiple devices.

Acknowledgments The authors would like to thank Ryo Yoshida for his support for this project.

References

1) Aoki, Y., Shinozaki, M. and Nakajima, A.: Creating Interactive Web Forms from XML Documents, *Proc. XML 2000*, pp.54–68 (2000).
2) Aoki, Y., Ando, F. and Nakajima, A.: Creating Web-based Presentations by Demonstra-

tion, *IPSJ Journal*, Vol.42, No.2, pp.155–165 (2001).
3) Aoki, Y.: Building a Collaborative Web Environment for Supporting End Users, *IPSJ Journal*, Vol.43, No.2, pp.530–542 (2002).
4) Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Hors, A.L., Nicol, G., Robie, J., Sutor, R., Wilson, C. and Wood, L.: *Document Object Model (DOM) Level 1 Specification Version 1.0*, W3C Recommendation (1998). Available at http://www.w3.org/TR/REC-DOM-Level-1/.
5) Barr, A. and Feigenbaum, E.: *The Handbook of Artificial Intelligence*, Volume I, Addison-Wesley, MA (1986).
6) Bergsten, H.: *Java Server Pages*, O'Reilly & Associates, MA (2000).
7) Boyer, J., Bray, T. and Gordon, M.: *Extensible Forms Description Language (XFDL) 4.0*, W3C Note (1998). Available at http://www.w3.org/TR/NOTE-XFDL/.
8) Bray, T., Paoli, J. and Sperberg-McQueen, C.M. (Ed.): *Extensible Markup Language (XML) 1.0*, W3C Recommendation (1998). Available at http://www.w3.org/TR/REC-xml/.
9) Clark, J.: *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation (1999). Available at http://www.w3.org/TR/xslt/.
10) Cloyd, M.H.: Designing User-Centered Web Applications in Web Time, *IEEE Software*, pp.62–69 (2001).
11) Cranor, L., Langheinrich, M., Marchiori, M., Presler-Marshall, M. and Reagle, J.: *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*, W3C Recommendation (2002). Available at http://www.w3.org/TR/P3P/.
12) Cypher, A. (Ed.): *Watch What I Do: Programming by Demonstration*, The MIT Press, MA (1993).
13) Dubinko, M., Dietl, J., Klotz, L.L., Jr., Merrick, R. and Raman, T.V.: *XForms 1.0*, W3C Candidate Recommendation (2002). Available at http://www.w3.org/TR/xforms/.
14) Dyche, J.: *The CRM Handbook: A Business Guide to Customer Relationship Management*, Addison-Wesley, MA (2001).
15) Fisher, G.L., Busse, D.E. and Wolber, D.A.: Adding Rule-Based Reasoning to a Demonstrational Interface Builder, *Proc. UIST '92*, pp.89–97 (1992).
16) Girgensohn, A. and Lee, A.: Seamless Integration of Interactive Forms into the Web, *Proc. 6th Intl. World Wide Web Conf.*, pp.1531–1542 (1997).
17) Goodman, D.: *Dynamic HTML: The Definitive Reference*, O'Reilly & Associates, MA (1998).
18) Graham, I.S.: *HTML Stylesheet Sourcebook*, John Wiley & Sons, NY (1997).
19) Grant, L.: Customer Service Shortfall Hits Net Sales, *USA Today* (June 1, 1999).
20) Gundavaram, S.: *CGI Programming on the World Wide Web*, O'Reilly & Associates, MA (1996).
21) Hartman, P., Bezos, J., Kaphan, S. and Spiegel, J.: Method and System for Placing a Purchase Order via a Communication Network, *United States Patent 5,960,411* (1999).
22) Hunter, J. and Crawford, W.: *Java Servlet Programming, 2nd Edition*, O'Reilly & Associates, MA (2001).
23) Kobayashi, M., Shinozaki, M., Sakairi, T., Touma, M., Daijavad, S. and Wolf, C.: Collaborative Customer Services Using Synchronous Web Browser Sharing, *Proc. CSCW '98*, pp.99–108 (1998).
24) Kristensen, A.: Formsheets and the XML Forms Language, *Proc. 8th Intl. World Wide Web Conf.*, pp.111–123 (1999).
25) Lieberman, H. (Ed.): *Your Wish is My Command: Programming by Example*, Morgan Kaufmann, CA (2001).
26) Lohse, G.L. and Spiller, P.: Electronic Shopping, *Comm.ACM*, Vol.41, No.7, pp.81–88 (1998).
27) Maruyama, H., Tamura, K. and Uramoto, N.: *XML and Java: Developing Web Applications*, Addison-Wesley, MA (1999).
28) Meyer, E.: *Cascading Style Sheets: The Definitive Guide*, O'Reilly & Associates, MA (2000).
29) Moriyon, R., Szekely, P. and Neches, R.: Automatic Generation of Help from Interface Design Model, *Proc. CHI '94*, pp.225–231 (1994).
30) Myers, B.A.: Creating User Interfaces Using Programming by Example, Visual Programming and Constraints, *ACM Trans.Prog.Lang. Syst.*, Vol.12, No.2, pp.143–177 (1990).
31) Myers, B.A. and Rosson, M.B.: Survey on User Interface Programming, *Proc. CHI '92*, pp.195–202 (1992).
32) Myers, B.A., McDaniel, R.G. and Kosbie, D.S.: Marquise: Creating Complete User Interfaces by Demonstration, *Proc. INTERCHI '93*, pp.293–300 (1993).
33) Nardi, B.A.: *A Small Matter of Programming*, The MIT Press, MA (1993).
34) Pixley, T.: *Document Object Model (DOM) Level 2 Events Specification Version 1.0*, W3C Recommendation (2000). Available at http://www.w3.org/TR/DOM-Level-2-Events/.
35) Ousterhout, J.K.: *Tcl and the Tk Toolkit*, Addison-Wesley, MA (1994).
36) Raggett, D., Hors, A.L. and Jacobs, I. (Ed.):

HTML 4.01 Specification, W3C Recommendation (1999). Available at http://www.w3.org/TR/html4/.

37) Shneiderman, B.: Direct Manipulation: A Step Beyond Programming Languages, *IEEE Computer*, Vol.16, No.8, pp.57–69 (1983).

38) Sukaviriya, P.: Dynamic Construction of Animated Help for Application Context, *Proc. ACM SIGGRAPH Symposium on User Interface Software*, pp.190–202 (1988).

39) Sukaviriya, P. and Foley, J.D.: Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help, *Proc. UIST '90*, pp.152–166 (1990).

40) Tai, H. and Yamamoto, G.: An Agent Server for the Next Generation of Web Applications, *Proc. 11th Intl. Workshop on Database and Expert Systems Applications (DEXA'00)*, pp.717–721 (2000).

41) Tanaka, Y.: IntelligentPad as Meme Media and Its Application to Multimedia Database, *Information and Software Technology*, Elsevier Science, Netherlands, Vol.38, No.3, pp.201–211 (1996).

42) Thistlewaite, P. and Ball, S.: Active FORMs, *Proc. the 5th Intl. World Wide Web Conf.*, pp.355–1364 (1996).

43) Tuck, R. and Olsen, D.R.: Help by Guided Tasks: Utilizing UIMS Knowledge, *Proc. CHI '90*, pp.71–78 (1990).

44) Yamamoto, G. and Nakamura, Y.: Architecture and Performance Evaluation of a Massive Multi-agent System, *Proc. Autonomous Agents '99*, pp.319–325 (1999).

45) Wells, N. and Wolfers, J.: Finance with a Personalized Touch, *Comm. ACM*, Vol.43, No.8, pp.31–34 (2000).

46) Wiecha, C., Bennett, W., Boise, S. and Gould, J.: Generating Highly Interactive User Interfaces, *Proc. CHI '89*, pp.277–282 (1989).

(Received April 30, 2002)
(Accepted November 5, 2002)

Yoshinori Aoki received the B.E., M.E., and Ph.D. degrees in Computer Science from Kyushu University, Fukuoka, Japan, in 1995, 1997, and 2003. In 1997, he joined Tokyo Research Laboratory, IBM Japan, Ltd. He has worked on Web-based interactive system designs in the laboratory. His research interests include human-computer interaction, XML, and distributed systems. He is a member of the ACM and the IEEE CS.

Masahide Shinozaki received the B.E. degree in information science in 1988 and the M.E. degree in information science and electronics in 1990 from the University of Tsukuba. He joined IBM Japan in 1990. He worked in IBM Research, Tokyo Research Laboratory from 1990 to 2000. He moved to IBM Global Services from March, 2000. His research interests include human-computer interaction and synchronous collaboration systems.

Amane Nakajima received the B.E. degree in electronic engineering in 1983 and the M.E. degree in electrical engineering in 1985 from the University of Tokyo. He joined IBM Japan in 1985. He has worked in IBM Research, Tokyo Research Laboratory for 15 years. Currently, he is the Managing Consultant in IBM Global Services. His research interests include human interaction systems and distributed systems. He received the Best Paper Award from the Institute of Electronics, Information and Communication Engineers of Japan in 1987. He is a member of the IEEE and the Association for Computing Machinery.