Model-driven development of large-scale Web applications

H. Tai
K. Mitsui
T. Nerome
M. Abe
K. Ono
M. Hori

This paper describes our approach to support the development of large-scale Web applications. Large development efforts have to be divided into a number of smaller tasks of different kinds that can be performed by multiple developers. Once this process has taken place, it is important to manage the consistency among the artifacts in an efficient and systematic manner. Our model-driven approach makes this possible. In this paper, we discuss how a metamodel is used to describe part of the specification as a central contract among the developers. We also describe a tool that we implemented on the basis of the metamodel. The tool provides a variety of code generators and a mechanism for checking whether view artifacts, such as JavaServer Pages™, are compliant with the model. This feature helps developers manage the consistency between a view artifact and the related business logic—HyperText Transfer Protocol request handlers.

1. Introduction

As Web applications grow in size, the size of development projects grows as well, and it becomes more and more critical to support modular application design and parallel development. The challenge lies in meeting a few unique requirements. This paper describes our model-driven approach to support the development of large-scale Web applications.

The first requirement to be met is that the solution support the appropriate division of the development effort into smaller and different kinds of tasks that can be performed by various kinds of developers, such as architects, screen designers, and business logic programmers. It must then support the efficient integration of the many outputs from these different tasks. There must also be a cost-effective method of unit and integration testing if we are to maximize the parallel progress of the tasks that are inherently dependent upon one other.

The second requirement is to support the integration of different kinds of programming technologies, such as Hypertext Markup Language (HTML), JavaServer Pages** (JSP**), Struts, JavaBeans**, and possibly other in-house technologies designed to best suit developers' specific but differing needs. The methodology for Web application development is still in its early stages, and new technologies continue to emerge. Sometimes these technologies are alternatives, and sometimes they are used in combinations. In the latter case, the binding between artifacts based on different technologies is loose. For example, representing names as character strings is still a popular way to represent interfaces or references that cross technology boundaries. However, since there is no standard way to guarantee the consistency of such references, they are one of the parts of Web applications that typically cause interface mismatch errors and are very error-prone. Such errors are costly to detect and correct if this must be done manually. As the size of Web applications grows, automatic consistency checking becomes critical.

To meet such requirements, we designed a metamodel[1] that describes the specification of a Web application, and we built a tool that utilizes the model (an instance of the metamodel). We use the specification description formalized by this metamodel as a central contract

[1] A model usually specifies a Web application in an abstract fashion. The metamodel for a class of models specifies how the models are specified.

0018-8646/04/$5.00 © 2004 IBM

among all developers on a project. To check that all of the artifacts produced by the developers are in compliance with the given specification, we use an automatic stub[2] code generation technique that performs the check in test runs. By checking this compliance, we can ensure consistency among different kinds of artifacts.

In an earlier paper [1], we discussed basic specification elements and associated support tools. This paper focuses on our model-driven solution, and we discuss in detail the effectiveness of the supported modeling elements and tools that were implemented to make use of the model description and facilitate Web application development efforts.

The paper is organized as follows. Section 2 discusses the preceding work related to our metamodel and tool. Section 3 reviews modern Web application development technologies and practices, which follow a specific methodology and are based on one of the most widely adopted Web application architectures—the model-view-controller (MVC) pattern [2]. Section 4 describes problems associated with current Web application development practices. In Section 5, we propose a model-driven development method as a solution to the problems identified in Section 4. Section 6 describes the metamodel for Web applications, which is the basis of the model-driven development described in this paper. The prototype tool we built on the basis of the metamodel is introduced in Section 7, and Section 8 concludes the paper.

2. Related work

The Unified Modeling Language (UML) [3] notation is used to model the business logic and server objects in Web applications. A UML extension [4] is proposed for describing Web applications in which the relationship between a form and a Web page is represented by a submit association stereotype between the Form and Server Page stereotyped classes. In terms of the relationship between a form and a Web page, our metamodel is similar to this UML extension. However, the model elements in this UML extension are not related to the components in the MVC pattern. For example, both views and business logic in the MVC pattern are represented by Server Page elements. Thus, it is not straightforward to use this UML extension for modeling Web applications that are based on the MVC pattern.

Another UML extension for describing Web applications [5] is also proposed. In its notation, page navigation is defined in a more structural way than in our notation. Users can design and maintain hyperlinks and data structure in a high-level model. In contrast, our notation is low-level, so that users should be aware of the MVC pattern.

HDM-lite is a hypermedia design model tailored to the development of Web applications; it is used to specify the structural, navigation, and presentation semantics for Web applications [6]. The HDM-lite model integrates database and hypermedia modeling concepts with presentation abstractions for the Web context. The navigation semantics in HDM-lite are prescribed on the basis of the movement of the focus from one page to another and are defined with presentation-related concepts, such as accessing, filtering, and indexing. In contrast, the navigation semantics in our metamodel are prescribed on the basis of the semantics of the uniform resource locator (URL) [7]. This difference in modeling perspectives stems from assumptions made about the intended scope of the Web applications. That is, HDM-lite assumes that Web applications are hypermedia-like and data-intensive, and that the central issue in the presentation is navigation. On the other hand, our metamodel assumes that Web applications with business transactions are dynamic, and that the navigation semantics must capture the invocation of server-side actions and the page flow.

A development environment called *Autoweb* [6], based on HDM-lite, has been created for data-intensive Web applications. Although Autoweb allows code generation and mapping from relational database schema to presentation pages on the Web, it assumes the common gateway interface (CGI) process as part of the runtime environment; therefore, it cannot be exploited, as it currently exists, for runtime frameworks based on the MVC architecture.

WebML [8] is a modeling language that exploits the entity relationship of data that is to be published by a Web application. It provides a sophisticated method of notation and a tool for designing Web applications for data retrieval and manipulation.

3. Structure of Web applications

In the early days of Web application development, applications did not have to be highly structured because they were simple and small. They were built using CGI technology, which means that a CGI program does everything needed to handle requests from HyperText Transfer Protocol (HTTP) clients. As applications became more complex and larger, the MVC pattern was widely adopted. In the MVC pattern, each request from a Web client is processed through the following steps: 1) interpret the request; 2) dispatch it to business logic; 3) select a view; and 4) generate the view content. In particular, on the Java** 2 Enterprise Edition (J2EE**) platform, it is recommended that the Model 2 architecture [2, 9] be used unless the application is small and simple.

[2] The term *stub* refers to a substitute for an artifact that is not yet completed but is needed for execution by other artifacts.

Figure 1 illustrates the Model 2 architecture, which is a realization of the MVC pattern on the J2EE platform. In this architecture, a component called the *front controller servlet* fills an important role. All requests from Web clients are first handled by the front controller servlet, which interprets the request and determines the operation to execute. Each operation is implemented as an Action component. An Action component may create model components or modify them through the execution of the operation. After that, the front controller servlet selects a View component, which is determined on the basis of the results returned from the execution of the Action. Finally, the selected View component generates the content to display and returns it to the Web client.

The primary benefit provided by the Model 2 architecture is that it makes Web applications easy to maintain through the decomposition of an application into a number of components whose roles are specialized. It is easier to develop individual components than to develop monolithic CGI programs, because developers need to know only technologies related to the components for which they are responsible, whereas CGI programmers are required to know every related technology (e.g., HTML, HTTP, scripting languages, and databases). For example, developers of View components must have knowledge of JSP and HTML. However, they do not have to deal with the interpretation of requests, the ways requests can be handled, page flows, or issues of database access.

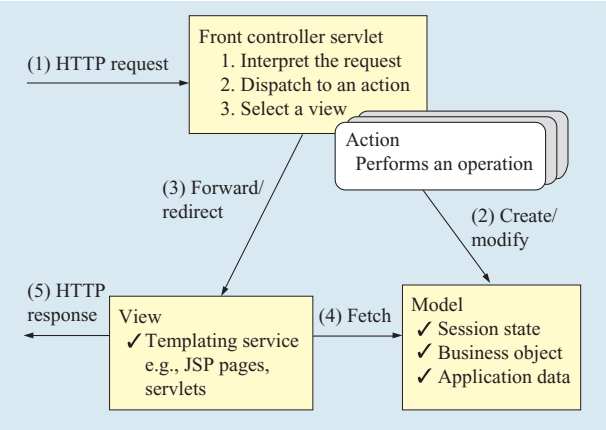

Figure 1

Model 2 architecture — a realization of the MVC pattern on the J2EE platform.

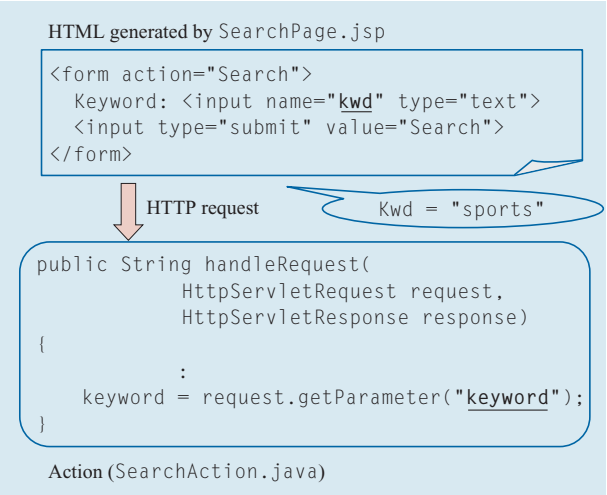

Figure 2

Sample of an inconsistency between a JSP and an Action.

4. Problems in modern Web application development

Web application development reveals some peculiarities and problems that stem primarily from two sources: the wide range of technologies involved and a lack of support for the division of work.

As Web applications become large and complex, they are built using various kinds of technologies. For example, in the case of J2EE platforms, they include HTML, JavaScript**, JSP files, Extensible Markup Language (XML), Enterprise JavaBeans** (EJB**), Remote Method Invocation (RMI), Java Database Connectivity (JDBC**), and so on. The sheer number of these technologies creates problems, and it can be quite difficult to find developers who know all of them. Consequently, modern Web applications are developed by various developers who have different skills.

As Web applications grow in size, solutions for scalable development become more important and necessary. One possible solution is to provide support for the division of design and implementation work and to assign the divided work tasks to a number of developers. To work efficiently, this solution has to manage the artifacts produced by those developers and make sure they are consistent with the specifications. However, there is no current scheme—such as a static-type system, to which the output from the divided work must conform—that can force all artifacts to be combined in a consistent manner.

Figure 2 illustrates a sample case in which the required consistency between a JSP and an Action is not satisfied. Referring to the figure, `SearchPage.jsp` contains an HTML form for sending a request to `SearchAction`. Although `SearchAction` expects that the HTTP request

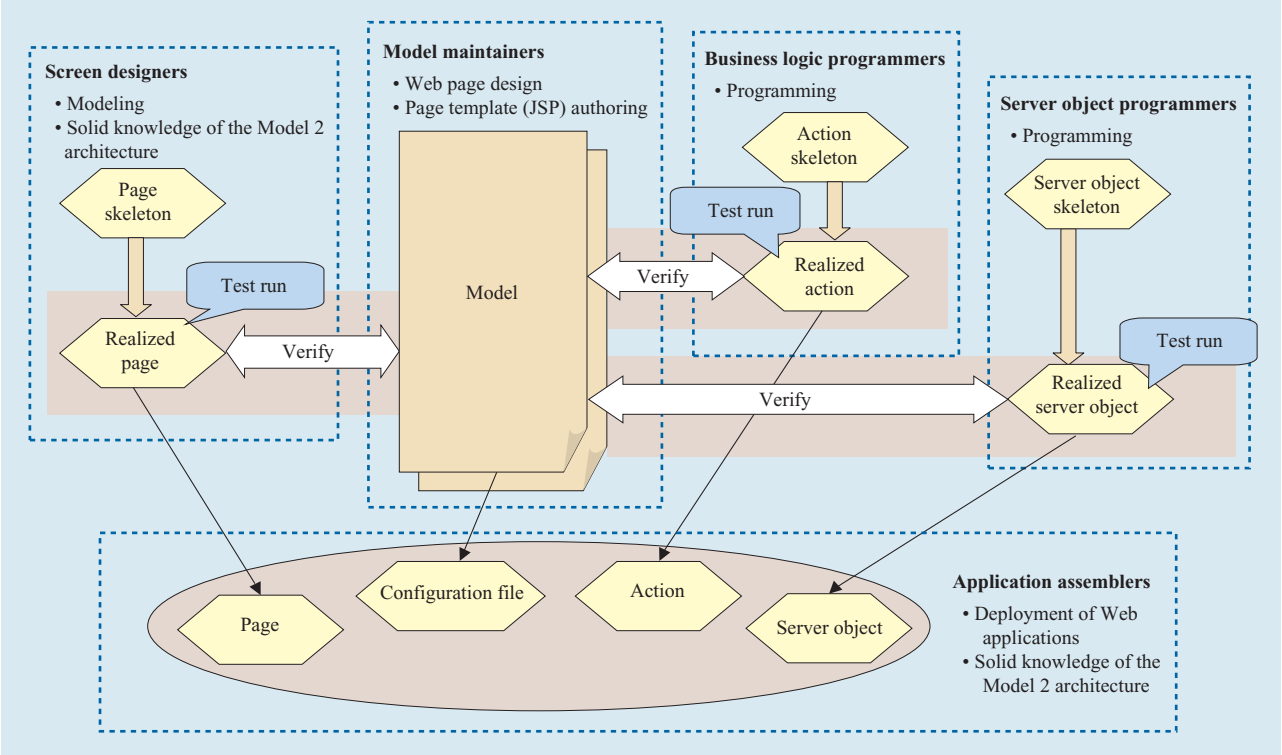

Figure 3

Model-driven development for Web applications.

contains the parameter named `keyword`, the HTML form in `SearchPage.jsp` sends out a request with a different parameter name, `kwd`. In this case, `SearchAction` behaves as if no search keyword was specified, even if a keyword was specified in the HTML form. The troublesome problem is that neither a warning nor an error is reported regarding this inconsistency, so it is quite challenging for developers to identify the source of the problem. The problem might be something in the search logic, an incorrect parameter name in the Action, or an incorrect field name in the JSP page. It would require unwelcome time and expense to fix the bug.

5. Model-driven development for Web applications

In this section, we describe a model-driven approach to solve these problems. **Figure 3** illustrates the development style that categorizes the developer roles and lists the developer roles and the skills required for each.

Screen designers

Screen designers are responsible for implementing View components, such as JSP pages and HTML files. The model developed by the model maintainer can be exploited to automatically generate skeleton code pages that are populated with the minimum page elements. The greater the number of View components that a screen designer has to develop, the greater the benefit of skeleton code generation. When a screen designer finishes implementing a View component, he has to verify that the implementation conforms to the model. For example, the model may have prescribed that the page must contain specific input and output fields.

Model maintainers

Model maintainers are responsible for creating and maintaining the model (or part of the model) that describes the outline of the portion of the Web application for which they are responsible. Because such a model usually becomes large, the model should be maintainable by multiple workers to deal with the scaling. Every time the model changes, its validity as a whole must be ensured, so software tools are required to perform this verification. The original specification might be created in the form of an unstructured document. In such cases, model maintainers are responsible for transforming the original specifications to a set of models.

Business logic programmers

Business logic programmers are responsible for implementing Action components. A skeleton code generator is provided for them as well. It can be much more beneficial in this role because programmers are able to understand the specification of each Action component in the form of program code. Like screen designers, programmers must verify that the implementation they develop conforms to the model.

Server object programmers

Server object programmers are responsible for implementing Model components, which are typically implemented as objects stored in an `HttpSession` object that is associated with each HTTP session. They may also be implemented as objects stored in a stateful session bean running on a J2EE server.

Application assemblers

Application assemblers are responsible for constructing a complete Web application by combining all of the artifacts developed by screen designers, business logic programmers, and server object programmers. They are also responsible for verifying whether the artifacts all conform to the model. This can be done by checking the verification results submitted by the developers. The verification results must be well maintained in order to ensure that they are really about the submitted version of artifacts. However, this is sometimes difficult, especially when the development teams are distributed among different locations or organizations. The surest verification is for the application assemblers to perform the checks themselves. However, because application assemblers must check so many artifacts each time at this stage, the checking should be automated.

To construct a complete Web application, application assemblers must prepare several configuration files. It is possible to generate most of the configuration files from the model.

6. Modeling Web applications

The central part of our solution is the metamodel for modeling Web applications. The Web application model described by the metamodel plays a role as an architectural blueprint, which describes the division of a large Web application into subapplications and provides the specification to which all artifacts in the development project must conform.

Our modeling scheme has historically been called *WAD* [1], which stands for *Web Application Descriptor*. WAD originally started as metadata that described the relationships among the artifacts of a Web application; it was used to automatically check consistency among the artifacts. It turned out, however, that such metadata constitutes more than just the relationships. Rather, it is the architecture of a Web application and should also include modules, useful abstractions, concerns about reuse, and so forth. Hereafter, in this paper, we call the metamodel the *WAD model*, or simply *WAD* when there is no possibility of confusion.

We comment here about Model-Driven Architecture (MDA) [10], which is an emerging industry standard for promoting active use of models in software development. There are applications of such model-driven approaches to Web applications [6, 11]. In MDA, focusing on platform independence is part of its value proposition, because making models platform-independent provides us with opportunities to systematically transform the models into various platform-dependent implementations. While our long-term intention is for MDA, WAD is dependent on the Model 2 architecture of J2EE [2, 6]. Determining the degree to which WAD is applicable to relatively different kinds of architecture is future work. This section describes the detailed modeling constructs used in WAD.

Pages and page transitions

Web applications provide the application front end, specifically, the actual presentation of a user interface and the control of user interaction flows. The application front end interacts with the application back end, such as databases and other kinds of system services, through standard Internet technologies, including Web browsers and HTTP. Thus, the application front end is about user interfaces.

The very first element to be modeled in a Web application—when the result of use case analysis is mapped onto the implementation model—is a collection of pages and the transitions between pages. A (Web) page is a unit of presentation composition. In a typical operating environment that serves Web applications, a page is prepared in a Web server and transmitted to a Web browser, which visually renders the page. The displayed pages usually include sections in the screen that trigger commands or data transmissions that cause actions to be taken at the server (HTML forms or hyperlinks). A trigger fires when a user clicks a hyperlink or a button in an HTML form, and the corresponding action results in the appearance of the next page. Thus, there are transition flows between pages. It is important to distinguish between the page as a unit of presentation at design time and the page that is actually prepared at runtime and transmitted to a Web browser, particularly when a design-time page is instantiated as a similar but slightly different page at runtime. In that sense, we call the former the *page template*, and the latter a *page instance*. We refer to a page template as simply a page, because this paper is concerned primarily with the time of design. When there is a

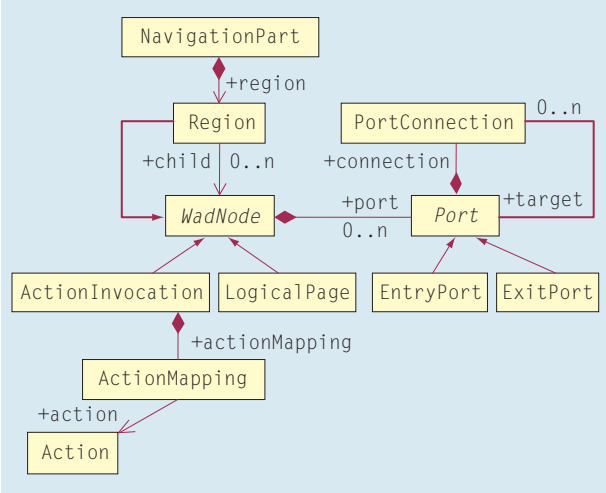

Figure 4

WAD metamodel elements related to page transition presented in the form of a class diagram in UML.

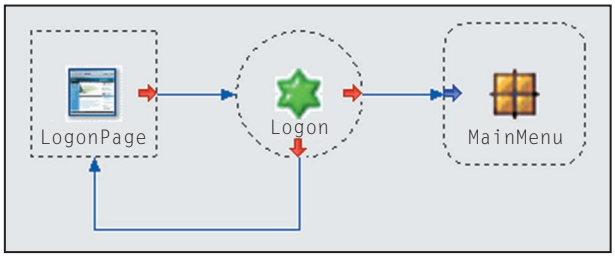

Figure 5

Sample graphical notation of a page transition model.

possibility of confusion, we use the term *page template*. A page template is a program in the sense that, at runtime, it instantiates page instances, which are marked-up documents transmitted to Web browsers.

Transitions between pages are modeled and represented as a directed graph. **Figure 4** shows the WAD metamodel elements related to page transition presented in the form of a class diagram in UML. A transition can be conditional; when an outgoing transition from a page is triggered, it is associated with data transmitted from the browser, which is typically user input. The transition flow may be determined by checking conditions based on the transmitted data. In WAD, this point of the conditional transition is called an *action invocation.* An action invocation refers to an *action* in which input parameters are specified. An *action* also specifies the possible result codes, and each result code is mapped to the next transition on the basis of the information specified by the action invocation. We present more details about the action element in the section below on data modeling.

Pages and transitions between pages are modeled as a directed graph of pages and action invocations. As the number of pages and action invocations increases, the corresponding directed graph becomes too complex to maintain. To keep directed graphs simpler, the notion of a *region* is introduced. A region contains a directed graph, while it is also regarded as a node. Page transitions can be modeled as a hierarchical directed graph that may include regions.

Another important part of the transition model is the notion of a *port.* Ports are attached to pages, action invocations, and regions. A port is either of an *entry* or *exit* type. An exit port can be the source of a link between two ports, and an entry port can be the target of a link. Ports attached to a region can be either the source or target of a link, and they are relay points of a transitional port connection between a port outside and another port inside the region. Generally, a chain of ports joins two artifact nodes, such as pages and action invocations. An entry port for a page or an action invocation respectively denotes an entry point to the page or the corresponding action.

In a specific implementation technology, such as J2EE, ports may be resolved as URLs representing pages and actions. The resolved URLs are embedded in Web application artifacts, such as pages and action programs, and used by the artifacts to refer to one another. It is very useful to model ports at this abstract level and to resolve ports automatically into URLs. The reason for this is that determining URLs requires careful consideration about the way in which pages and actions are physically deployed, and they are likely to change depending on nonfunctional considerations, such as security and access controls, that happen to be tied to the file directory structure. Ports hide the unnecessary details of such referential information that is dependent on implementation technology, but provide sufficiently detailed relationships between pages and action invocations.

Figure 5 illustrates our notation for pages, action invocations, and ports. This sample is composed of three nodes: a page, `LogonPage`, an action invocation, `Logon`, and a region, `MainMenu`. Exit ports are illustrated as thick red arrows in `LogonPage` and `Logon`. An entry port is shown as a thick blue arrow in `MainMenu`.

In most cases, a page is assigned a URL for requests of page content, and an action invocation is assigned a URL for invocations of the action, including its business logic. Regions are not mapped to any implementation resources.

Structured page content

The content of a page is often partitioned and grouped into multiple sections. For example, in some cases, pages are composed of a set of HTML frames; in other cases, pages are generated using a page template that includes other subtemplates, which generate HTML fragments. Because we need the ability to model a structure that expresses HTML frames or page templates that consist of subtemplates, we introduce the notion of a *frameset* and *frame*.

Figure 6 illustrates a sample graphical notation of a page flow with a frameset and frame. A *frameset* defines a page template that is composed of one or more subtemplates. A *frame* defines a subtemplate, which occupies a partition within a frameset. In terms of hierarchical page flows, a frameset is considered a unit of presentation, e.g., a URL that contains a page flow within each child frame. The model illustrated in Figure 6 declares that

1. The frameset FS consists of frames LEFT and RIGHT.
2. Pages NaviA and NaviB appear in the frame LEFT.
3. Pages B1, B2, and B3 appear in the frame RIGHT.

When a user activates (clicks) the exit port in the page Welcome, which is connected to frameset FS, the frameset FS is displayed in the browser window. The frameset FS displays a page that consists of two frames: LEFT and RIGHT. The entry point of a frameset, which is illustrated as a yellow dot in the figure, specifies the initial pages to be displayed in each frame. The frames might be implemented as HTML frames or HTML fragments embedded in the page. If a user activates an exit port in page B1, the contents in the frame RIGHT are replaced with the one generated by page B3 because the exit port is connected to page B3. If a user activates an exit port in page B3, the frameset is reset, and a plain Web page, Bye, is displayed, because the exit port is linked to page Bye located outside of frameset FS. A diamond represents an interframe page transition, called a *jumper-out*. A jumper-out is connected to a *jumper-in,* which is illustrated as a circle. If a user activates a jumper-out from page NaviA, page B1 is displayed within the frame RIGHT, while page NaviA is still displayed in the frame LEFT.

The notion of a frameset and frame in WAD can also be used to describe the inclusion of page templates. A frameset represents a page template (JSP) that includes other page templates, and a frame represents a placeholder within the container page template (frameset). Each page within a frame appears at the place specified by the frame.

Data model

In addition to triggers (e.g., HTML links and buttons) for page transitions, a page often includes user interface controls that receive user input. A set of input data that

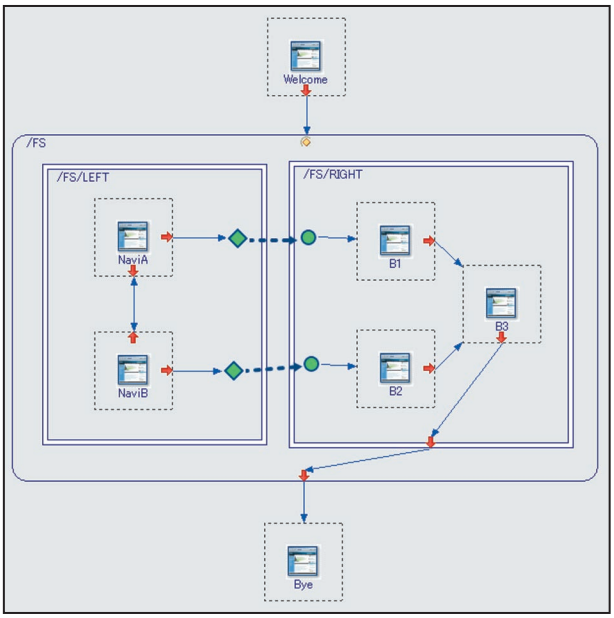

Figure 6

Sample graphical notation of a page flow with a frameset and frame.

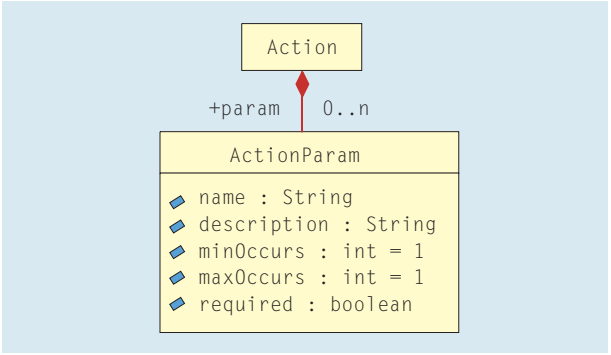

Figure 7

WAD metamodel (`ActionParam`) in the form of a class diagram in UML.

is transferred to an action may be further transferred to back-end applications or database systems, it may be temporally stored for later use in the application session, or it may just be used to determine the next transition. As we noted in Section 4, bugs that are hard to detect can result from misunderstandings by developers and protocol mismatches between input data sent from a page and the expectations of the actions that receive the data. Therefore, it is worthwhile to specify precisely the properties of data in the model and force all related artifacts to comply with it.

Figure 7 shows the *action parameter* (`ActionParam`) element of the WAD metamodel in the form of a class

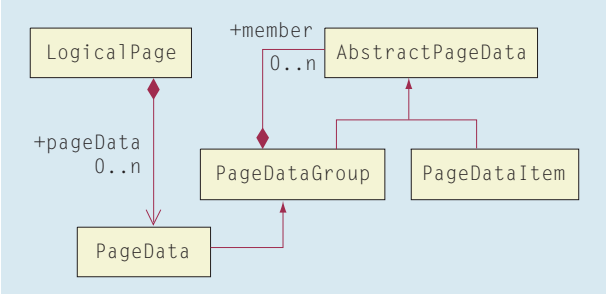

Figure 8

WAD metamodel (`PageData`).

diagram in UML. The action parameter specifies the input data expected by the action. It also specifies the input field that must be embedded in the page that invokes the action. Each action parameter is identified by its `name` attribute. The `required` attribute specifies whether or not the parameter (input data) is required for the action. The `minOccurs` and `maxOccurs` attributes specify the multiplicity of the data, i.e., array or scalar.

In addition to the data transferred to actions, we may want to prescribe in the model what dynamic data should be displayed in a page. The notion of *PageData* is introduced in the WAD metamodel to describe such information. As illustrated in **Figure 8**, a `LogicalPage`, or simply a page, can have zero or more `PageData` elements, each of which specifies the schema of an object that is available in the page for data access. The `PageData` elements form a tree structure whose leaves are of the type `PageDataItem`. A `PageDataItem` represents a scalar value used in the page data, whereas the `PageDataGroup` represents the recursion of the structure.

Scalability of modeling

As the number of pages and action invocations increases, it becomes difficult for a single person to capture and maintain the model. Therefore, it should be possible for a model to be developed and maintained by many people working in parallel. To divide a WAD model into several submodels, we introduced the *region reference*. A region reference looks like a region, except that it is just a reference to a region. By having region references in a model, a designer can change the elements to which those references refer without affecting other parts of the model that use the references. This is particularly useful when some regions are not yet modeled in detail, but other parts that refer to those regions are adequately modeled and the region interfaces are verified. It may also be useful when stubs that mimic the missing regions are available. The completed regions are integrated later. Note that with modular modeling, programmers can work in teams during the implementation phase of projects, making parallel development possible.

Security

Pages or action invocations sometimes have to be requested through HyperText Transport Protocol (Secure) (HTTPS). This is specified as the HTTPS attribute of pages or action invocations in WAD models. A page or an action invocation whose HTTPS attribute is set to true signifies that it must be requested through HTTPS. If the HTTPS attribute of a region is set to true, all nodes (i.e., pages and action invocations) within the region are treated as if their HTTPS attributes are set to true. This part of the model can be exploited when our program generation tool determines the URL of each node. No other current security-related modeling elements, such as authentication, are yet provided, because they may depend on specific runtime environments.

Annotating the model

While the WAD metamodel is designed independently of specific runtime frameworks, a model whose runtime-independent part is specified in WAD also requires runtime-dependent information, since it eventually has to be a Web application built within a specific runtime framework. For example, when an action is invoked in Struts, a model must specify the class name of the object in which the action parameters are stored. This information is required to generate the configuration files for Struts applications, but this does not make sense for other runtime frameworks. In a WAD model, this kind of platform-dependent information is described as *annotation* objects. Annotation objects can be attached to most WAD model elements and treated as an additional attribute of the element to which they are attached.

7. Tools and an integrated development environment

In this section, we describe a tool prototype called the *Web application development support tool* (hereafter, *WAST workbench*). It was built to facilitate and exploit the WAD metamodel. The WAST workbench is implemented as a set of plugins for the WebSphere* Studio platform [12], which is based on Eclipse technology [13]. **Figure 9** shows a screenshot of this tool. The WAST workbench consists of several views and an editor. WAD files are displayed in the navigator view, (a) in the figure, which is a built-in view provided by the platform. When a WAD file is opened by the editor and is selected, a specific menu item for this editor is added to the main menu. An example is shown in (b), and the model information is displayed in the editor and in two other views. The page flow editor (c)

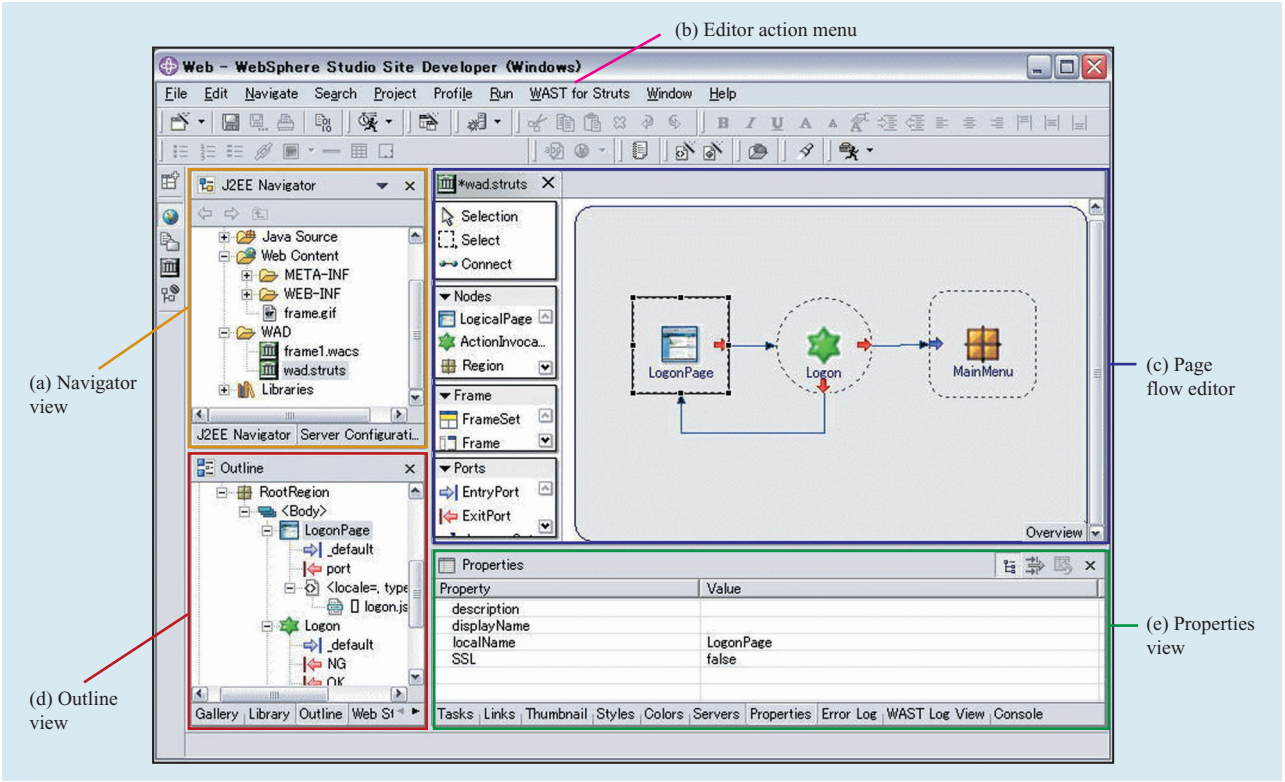

Figure 9

Screen shot of WAST workbench.

enables users to edit the page transitions as a graph. The outline view (d) shows all of the model elements as a tree, and users can add or remove model elements in this view. The properties view (e) displays the properties of the model element selected in the outline view or the page flow editor.

The WAST workbench consists of base plugins and an extension plugin. The base plugins provide the core functions, such as model manipulation, customizable implementations of the editor, and the views. The extension plugin provides concrete implementations of the views and the editor, along with the editor actions that support the development roles described in Section 5. **Figure 10** depicts the functions we have implemented for the Struts runtime environment: the model validator, the skeleton code generator, the configuration file generator, and the stub code generator.

Model validator

This function investigates models and detects errors, such as missing model elements, missing or inappropriate attribute values, and disconnected ports. It helps model maintainers ensure the correctness of models, which is also important for various code generators as well.

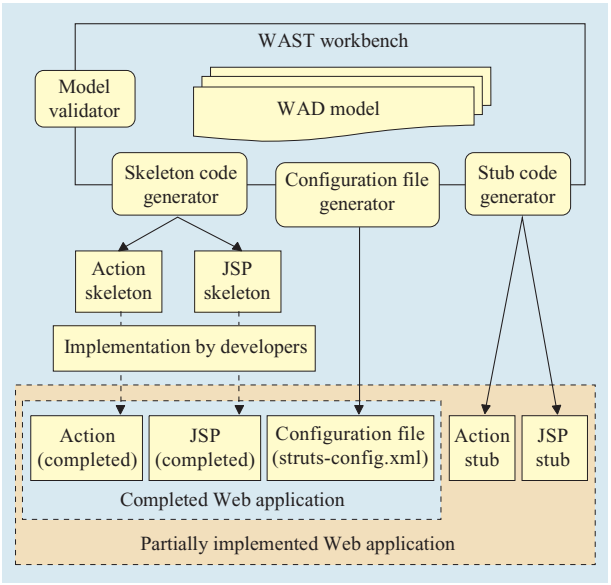

Figure 10

WAST workbench generator functions implemented for the Struts runtime environment.

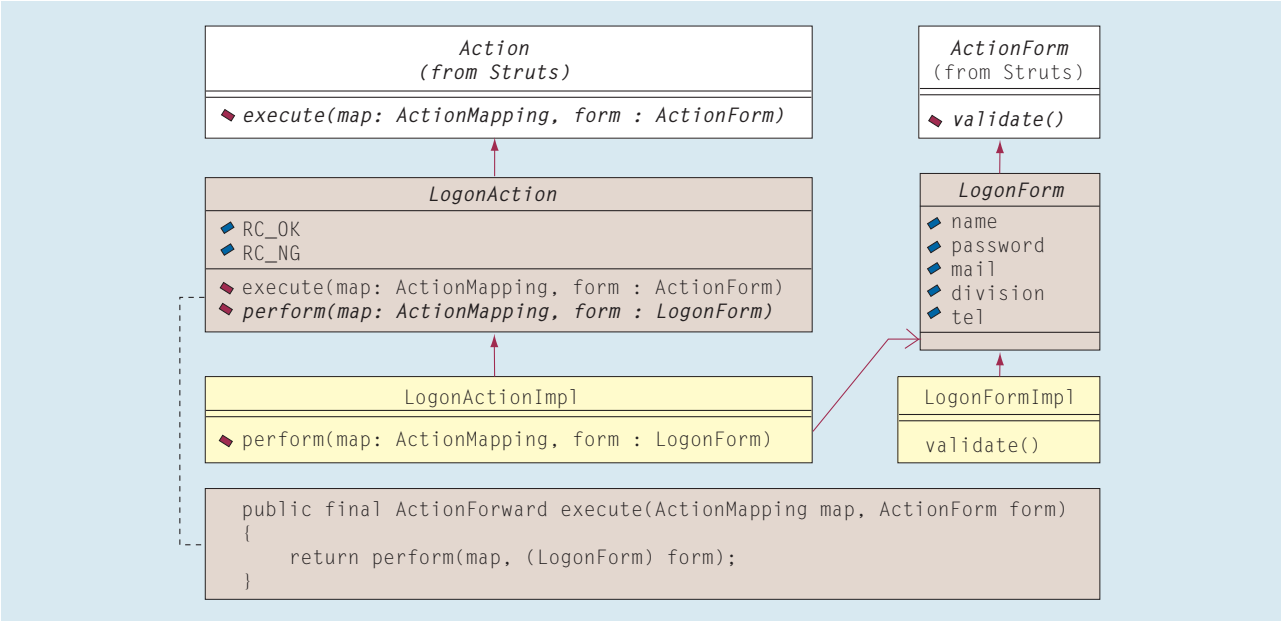

Figure 11

Class hierarchy of the generated classes. Abstract methods are denoted in italics.

Skeleton code generator

It is beneficial if we can have a good starting point for the implementation work, especially when the number of artifacts is large. Our tool prototype generates three kinds of skeleton code for Struts applications from WAD models:

1. A JSP file from a page in the model.
2. An action class from an action invocation in the model.
3. A bean class from the page data in the model.

For example, a skeletal JSP file contains a title, a description, static hyperlinks, HTML forms to invoke actions, and page data (output items).

One typical problem related to the code generator is how to update the generated code as the model changes. Regarding action class generators and bean class generators, we solved this problem by generating abstract classes that are to be extended by developers. Each time the model is changed, the tool generates the abstract classes. Developers are notified—in the form of compile errors—of the need to modify the implementation classes when they recompile the implementation classes.

Figure 11 shows a class hierarchy of the classes related to a generated class of a Struts application. The `Action` class and the `ActionForm` class are provided by Struts. `LogonAction` and `LogonForm` are the abstract classes generated by the code generator. `LogonActionImpl` and `LogonFormImpl` are the classes that are implemented by developers. The generated `LogonAction` class contains the constant fields of the result codes, and the definition of the *execute* method and the declaration of an abstract method named *perform,* which is introduced by the generator. If we change the name of the result code from OK to SUCCESS, the constant field, `RC_OK`, is replaced by `RC_SUCCESS`; the references to `RC_OK` then cause compile errors (references to an unknown field). It is not difficult for developers to modify such source code.

Configuration file generator

For Struts, developers must prepare a configuration file, named *struts-config.xml,* in which the page flows are described and other information is given. Although this method is better than embedding page flow information in various blocks of program code, it is still difficult to keep the configuration file correct. Our tool prototype can generate a complete set of page flow descriptions from a valid model.

Stub code generator

Our tool prototype generates stub code for pages and actions. Each developer can construct an executable Web application (partially implemented Web application) that is a mixture of completed artifacts and generated stub code. Action stub code checks incoming requests and reports any inconsistency in the requesting page (JSP) built with the WAD model. **Figure 12** shows a dialog

window where an automatically generated stub action pops up and reports an error in the requesting page. Screen designers can test whether or not a JSP can be executed, then whether or not the response is correct, and finally whether or not the JSP sends out the correct request for the subsequent action. In Figure 3, to the left of the model, these checks correspond to the arrow labeled "Verify" that is drawn between the model and the realized page.

As described in Section 4, Web application development involves two problems: the wide range of technologies and a lack of support for the division of work. To tackle the variety of technologies, we defined a set of roles and their activities as described in Section 5. In this case, it is important to localize the varieties of required technologies for each role. The MVC pattern or the Model 2 architecture makes it easy to divide the total work into tasks for each development role (screen designer, business logic programmer, or server object programmer). However, the MVC pattern or Model 2 architecture cannot help us maintain consistency among the artifacts created by each type of developer. To maintain consistency, developers require the specification of the application they are going to develop. Our metamodel provides a way to explicitly describe the basic portion of the specification. In addition, the tool prototype assists developers with their tasks by providing various generators and the means to check the conformity to the model.

8. Conclusion

We started with the observation that a single Web application development project typically uses a mixture of various technologies, and this makes debugging and maintenance difficult because of the absence of tool support to manage consistency among the various artifacts. The model-driven development discussed in this paper provides a way to design and arrange different kinds of artifacts in a consistent and cohesive way. Our metamodel for Web applications and our tools facilitate the work of developers by providing a model validator for model maintainers, skeleton code generators, stub code generators, and a configuration file generator. In particular, a generated action stub can check whether the requester page conforms to the model information. This kind of conformance check is important to the model-driven development for Web applications described in this paper because it reduces interface mismatch errors among the artifacts.

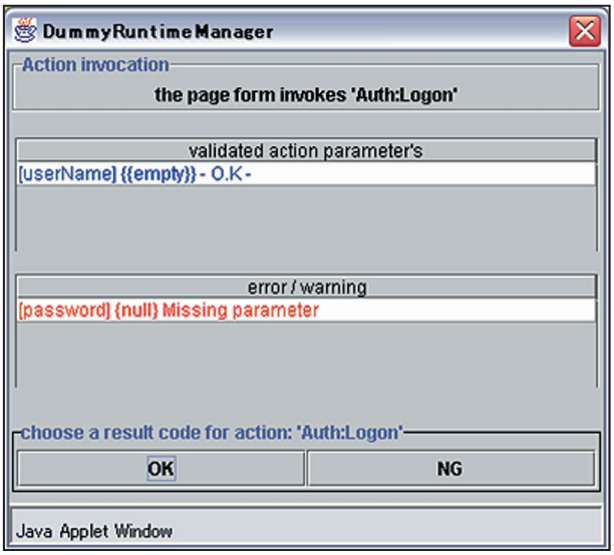

Figure 12

Action stub dialog window reporting missing parameter.

Acknowledgments

The authors acknowledge the following individuals for their support for this work and for their technical discussions: T. Watanabe, K. Hosokawa, N. Makoto, and T. Kamimura. The authors also received valuable comments and support for this work from the following individuals: H. Ida, T. Kanazawa, Y. Yashiro, D. Takamizawa, S. Ishii, N. Matsukage, and A. Sakakibara. The authors also wish thank to K. Kuse and Y. Takao for their continuous management support of our research.

References

1. H. Tai, T. Nerome, M. Abe, and M. Hori, "Model-Driven Development of Dynamic Web Applications," *Proceedings of the Conference on Extreme Markup Language,* 2002; see *http://www.idealliance.org/papers/extreme03/html/2002/Hori01/EML2002Hori01-toc.html*.
2. I. Singh, B. Stearns, M. Johnson, and the Enterprise Team, *Designing Enterprise Applications with the J2EE Platform,* Addison-Wesley Publishing Co., Reading, MA, 2002; ISBN: 0201787903.
3. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, First Edition, Addison-Wesley Publishing Co., Reading, MA, 1998; ISBN: 020130998X.
4. J. Conallen, "Modeling Web Application Architectures with UML," *Commun. ACM* **42,** No. 10, 63–70 (October 1999).
5. N. Koch, A. Kraus, and R. Hennicker, "The Authoring Process of the UML-based Web Engineering Approach," *Proceedings of the 1st International Workshop on Web-Oriented Software Technology*, 2001; see *http://www.dsic.upv.es/~west2001/iwwost01/IWWOSTContent.htm*.
6. P. Fraternali and P. Paolini, "Model-Driven Development of Web Applications: The Autoweb System," *ACM Trans.*

Office Info. Syst. **18,** No. 4, 323–382 (2000).
7. Uniform Resource Locators (URL): RFC 1738; Internet Engineering Task Force (IETF); see *http://www.ietf.org/rfc/rfc1738.txt*.
8. S. Ceri, P. Fraternali, and M. Matera, "Conceptual Modeling of Data-Intensive Web Applications," *IEEE Internet Computing* **6,** No. 4, 20–30 (July/August 2002).
9. G. Seshadri, "Understanding JavaServer Pages Model 2 Architecture: Exploring the MVC Design Pattern," *Java World,* December 1999; see *http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html*.
10. Model Driven Architecture (MDA), OMG Document ormsc/2001-07-01 edition, July 2001; Object Management Group, Inc., 250 First Ave., Suite 100, Needham, MA 02494; see *http://www.omg.org/*.
11. S. Ceri, P. Fraternali, and A. Bongio, "Web Modeling Language (WebML): A Modeling Language for Designing Web Sites," *Computer Networks* **33,** No. 1–6, 137–157 (2000).
12. IBM Corporation, IBM Redbooks, *WebSphere Studio V5 Overview and Architecture* (2003); see *http://www.redbooks.ibm.com/redpapers/pdfs/redp3742.pdf*.
13. Eclipse.org Consortium, Eclipse Platform, 2002; see *http://www.eclipse.org/*.

Received October 17, 2003; accepted for publication February 4, 2004; Internet publication September 17, 2004

Hideki Tai *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (hidekit@jp.ibm.com).* Mr. Tai received a B.S. degree in information sciences and an M.S. degree in information sciences and electronics from the University of Tsukuba, Japan, in 1995 and 1997, respectively. He joined the IBM Research Division in 1997, and his research projects have included mobile agents, a high-performance agent platform, and a Web application development tool. Mr. Tai's current research interests include software development tools and process, application framework, and middleware.

Kinichi Mitsui *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (mitsui@jp.ibm.com).* Mr. Mitsui joined the IBM Research Division in 1985 after receiving B.S. and M.S. degrees in information science from the Tokyo Institute of Technology. He began work on medical information systems and, until the present, worked primarily on software development tools and object-orientation technologies. He is currently the Senior Manager of the Software Technology Department at the Tokyo Research Laboratory. Mr. Mitsui's current interests include rapid software development tools, application frameworks for interactive software, and model-driven software development. He is a member of the Association for Computing Machinery (ACM).

Takashi Nerome *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan (nerome@jp.ibm.com).* Mr. Nerome received B.S.E and M.S.E. degrees in information engineering from Ryukyu University, Japan, in 1994 and 1996, respectively. He joined IBM Japan Ltd. in 1996 and worked as a systems engineer, mainly producing Web application systems. He subsequently qualified as an I/T specialist. Mr. Nerome has been working on methods and tools for Web applications in the IBM Research Division since 2000. He is a member of the Information Processing Society of Japan.

Mari Abe *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (maria@jp.ibm.com).* Ms. Abe received a B.E. degree in electrical engineering and an M.E. degree in computer science from Keio University in 1998 and 2000, respectively. Her research interests include Web content authoring and Web application development tools. Ms. Abe currently works at the Tokyo Research Laboratory; she is a Ph.D. candidate in the School of Science for Open and Environmental Systems, Graduate School of Science and Technology, Keio University.

Kohichi Ono *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (onono@jp.ibm.com).* Mr. Ono joined the IBM Research Division in 1994 after receiving B.S.E. and M.S.E. degrees in electrical engineering from Waseda University, Japan, in 1987 and 1989, respectively. He was a research associate at Waseda University from 1990 to 1992. He has worked on an impact analysis tool of COBOL programs, an online form application framework for medical information systems, mobile agent framework Aglets, XSLT stylesheet generation by demonstration, and Web application

development technologies. Mr. Ono's research interests include formal development methods, object-oriented application frameworks, software development tools, and mobile agent programming. He is a member of the IEEE Computer Society.

Masahiro Hori *Faculty of Informatics, Kansai University, 2-1-1 Ryozenji-cho, Takatsuki-shi, Osaka 569-1095, Japan (hormi@res.kutc.kansai-u.ac.jp).* Dr. Hori received a B.S. degree in biophysical engineering and M.E. and Ph.D. degrees in computer science from Osaka University, in 1984, 1986, and 1989, respectively. His focus was on natural language processing for speech understanding systems. In 1989 he joined the IBM Tokyo Research Laboratory, where he spent the next 14 years working on knowledge engineering methodologies, object-oriented software reuse, and Web content authoring and adaptation. Dr. Hori became a professor of informatics at Kansai University in 2003. His current interests lie in the universal design of Web information and the model-driven software development method. He received Research Awards in 1992 and 1997 from the Japanese Society for Artificial Intelligence. Dr. Hori is a member of the Web Ontology Working Group at the World Wide Web Consortium (W3C).