A Flexible and Efficient Presentation-Architecture for Adaptive Hypermedia: Description and Evaluation*

Carsten Ullrich, Paul Libbrecht, Stefan Winterstein, Martin Mühlenbrock
German Research Center for Artificial Intelligence
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
dev@activemath.org

Abstract

Means of achieving personalization in the Web are Adaptive Hypermedia techniques and the flexible composition of learning content from individual learning objects. If the objects are represented in a format different from the presentation format (e.g. as XML as opposed to HTML), a potentially expensive transformation process is required. Furthermore, caching becomes impractical depending on the degree of personalization, as the dynamic information is available at presentation time only and varies for each user. In this paper we present a flexible and efficient presentation pipeline based on a Model-View-Controller architecture, which overcomes performance problems and couples caching and personalization. It transforms single learning objects and uses a view-layer for adaptive presentation and navigation support that provides additional advantages such as incremental rendering and the easy adaption to different layout styles. Both theoretical and simulation results prove the efficency of this architecture.

1 Motivation

To detect and address a learner's needs is key to successful teaching. In Web-based systems, the responsiveness to the individual context is called Adaptive Hypermedia (AH).

Various techniques for achieving personalization in Web-based systems exist. In [1], Brusilovsky coined the terms "adaptive presentation" (e.g., text fragments on a fixed page are hidden or displayed), "adaptive navigation support", (e.g., annotating links with information about the knowledge state of the user), and "adaptive content selection" (a system selects and sorts content items).

Recently, the development of AH techniques has been influenced by the Semantic Web. As a result, the importance of encoding the learning material in a more abstract representation than HTML, for instance XML, has been recognized. The advantages are numerous. First of all, the rendering of different output formats, e.g., HTML and printer-friendly PDF becomes easily possible. What's more, by removing presentational information the reuse of learning material is eased, a factor that substantially reduces the total cost of authoring content. By referring to a DTD or XML-Schema, the learning material can be structured semantically. OMDoc ([2]) for instance, defines a knowledge representation targeted at representing mathematical documents at the text fragment level, thus providing a way to mark areas as an example or a definition. In this way, authors can exchange learning material at a fine-grained level and moreover intelligent learning systems can automatically generate courses from these learning material.

However, representing content in a format different from the output format requires a transformation process at presentation time, such as XSLT for XML. Depending on the amount of AH techniques involved, the transformation process requires substantial resources, both regarding CPU power and PC memory.

ActiveMath ([3]) for instance, a dynamic and adaptive web-based learning environment, composes an individual course for each learner. The course depends on the learner's goals, preferences and capabilities. Each page can consist of various learning material in varying order, which can be annotated differently. Figure 1 shows an example of an exercise session. Clearly visible is a text fragment, in this case an exercise. When a learner requests a detailed guided tour, ActiveMath generates a course with a page consisting of an introduction and a definition, as well as several examples and exercises.

In principle, this dynamic and adaptive composition of a course requires a complete transformation of the knowledge representation to the output format each time a learner accesses a page. In fact, the first version of ActiveMath reiterated the complete presentation cycle at every request.

At first glance, this transformation process cannot be op-

*This publication was generated in the LeActiveMath project, funded under FP6 (Contr. N. 507826). The authors are solely responsible for its content, it does not represent the opinion of the EC, and the EC is not responsible for any use that might be made of data appearing therein.

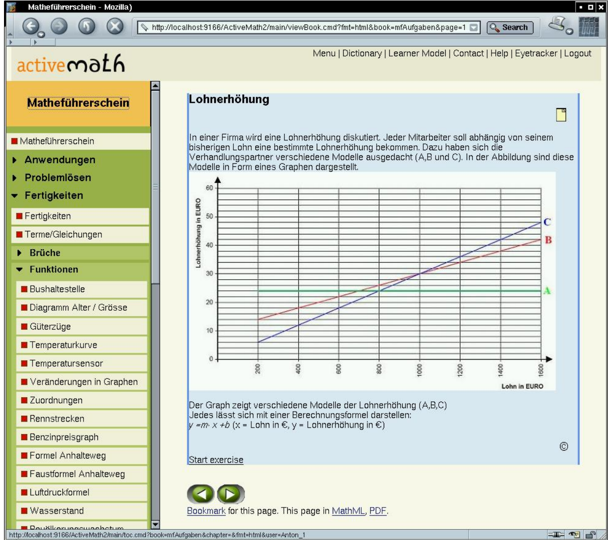

Figure 1. Screenshot of a page in ActiveMath.

timized. Caching a complete page would not help: Because of the dynamic features it is unlikely that the very same page will be shown to different users. Caching single fragments is not a solution either, as the single fragments are annotated using individual information available only at request time.

In this paper, we describe a presentation pipeline based on a Model-View-Controller architecture that overcomes these problems (Section 2). It transforms single fragments and uses the view-layer for realizing AH techniques. The view layer provides additional advantages such as incremental rendering and the easy adaption for different layouts. Both theoretical and simulation results prove the efficiency of this architecture (Section 3).

2 Description of the Architecture

A Model-View-Controller (MVC) architecture serves as the basis for our web application. It separates application logic (*controller*) from application data (*model*) and the presentation of that data (*view-layer*). It is well established for Java web applications, where servlets act as controllers, Java data objects form the model, and views are usually implemented by a dedicated template language.

The web application part of ACTIVEMATH is based on two frameworks: MAVERICK and VELOCITY. MAVERICK (`http://mav.sourceforge.net/`) is a minimalist MVC framework for web publishing using Java and J2EE, focusing solely on MVC logic. It provides a wiring between URLs, Java controller classes and view templates.

VELOCITY (`http://jakarta.apache.org/velocity/`) is a high-performance Java-based template engine, which provides a clean way to implement the view-layer and incorporate dynamic content in text based templates such as HTML pages. It provides a well focused template language with a powerful nested variable substitution and some basic control logic.

2.1 The Presentation Pipeline

We developed a 2-stage approach for presentation that uses XSLT transformations combined with a template engine. Basically, the presentation pipeline, as shown in Figure 2 can be divided into two stages.

The first stage deals with individual content fragments or items, which are written in XML and stored in a knowledge base. At this stage, content items do not depend on the user who is to view them. They hold unique IDs and can be handled independently. It is only in the second stage that items are combined to user-specific pages and enriched with dynamic data for this specific page request.

In Stage 1, the first part of the presentation pipeline is comprised of the steps *fetching, pre-processing*, and *transformation*. The *fetching* step collects requested content from the knowledge base. The output of this step are XML fragments. During *pre-processing* server-specific information is inserted into the XML content. Then the *transformation* performs the conversion into the output format by applying an XSLT stylesheet to the document. The output of the last step are text-based content fragments in, e.g., HTML or LaTeX. To summarize, Stage 1 delivers XSLT-transformed content fragments in the required output format. In Stage 2, these fragments are composed into a complete page and enriched with dynamic data for this specific page request, thereby allowing for AH techniques.

Stage 2 performs the steps *assembly, personalization* and optionally *compilation*. The *assembly* joins the fragments together to form the requested pages. The fragments in the desired output format are integrated into a page template, which is fetched from an external source. During the *personalization*, request-dependent information is used to add personalized data to the document, such as user information, knowledge indicators. If necessary, the *compilation* applies further processing to convert the generated textual content presentation into a binary format such as PDF (generated from LaTeX).

2.2 Example

The following simplified example illustrates the steps of the presentation pipeline. Say user Anton requests an HTML page that contains only one content item, "Definition 1" with the ID `def1`.

In the first step, this content is *fetched* from the knowledge base, which returns an OMDoc XML fragment:

```
<definition id="def1">
 Definition 1 with a reference to
 <ref xref="def2">Definition 2.</ref>
</definition>
```

The *pre-processing* step replaces the IDs of the items. Here, it adds the name of the knowledge base:

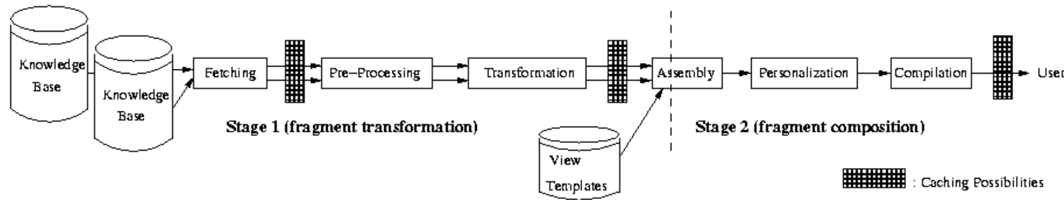

Figure 2. The presentation pipeline.

```
<definition id="kb1://def1">
 Definition 1 with a reference to
 <ref xref="kb1://def2">Definition 2.</ref>
</definition>
```

Then the fragment is *transformed* to HTML by using XSLT:

```
<div class="definition" id="kb1://def1">
 Definition 1 with a reference to
  $link.dict("Definition 2", "kb1://def2").
</div>
```

We see that the content item has been wrapped with an HTML `div` tag, and that the reference to Definition 2 has been replaced by `$link.dict()`. This is a variable reference for the view layer, and will later result in a call to a special Java helper bean that will generate the desired HTML code for this link.

In the *assembly* step, the items are put together to form a complete HTML page. Here, we only have a single content item, which is embedded in an HTML template:

```
<html> <!-- page template -->
 <head/>
 <body>
  This page is generated for $user.Name.
  <!-- begin item -->
  <div class="definition" id="kb1://def1">
   Definition 1 with a reference to
   $link.dict("Definition 2", "kb1://def2").
  </div>
  <!-- end item -->
 </body>
</html>
```

In the final step, the document is interpreted by VELOCITY. All variable references are resolved and replaced by the correspoding text, which is then sent to the user's browser:

```
<html> <!-- page template -->
 <head/>
 <body>
  This page is generated for Anton.
  <!-- begin item -->
  <div class="definition" id="kb1://def1">
   Definition 1 with a reference to
   <a onClick="openInDictionary('kb1://def2')">
    <!-- knowledge indicator -->
    <img src="green.jpg"/>
    Definition 2
   </a>.
  </div>
  <!-- end item -->
 </body>
</html>
```

2.3 The View-Layer

As we saw in the example, the VELOCITY template engine plays a central role in Stage 2. Its major function is to replace variable references with dynamic data that is only available at request time.

This data and the names under which it is available to the view-layer of the MVC architecture as VELOCITY a template is defined in our view layer interface specification. This document describes what data objects ("beans") can be accessed in each view and the properties they expose. Therefore, our XSLT stylesheets can output "intermediate data" in the form of variable references, which will later be replaced by actual dynamic data.

Without the use of a template engine, dynamic data would have to be available to the XSLT stylesheet at transformation time. Among other problems (such as making the XSLT stylesheet very much dependent of the HTML layout), this would make the caching of transformed content impractical, since transformed content would be different from user to user.

2.4 Incremental Rendering

In order to improve the perceived performance of our application, we chose an incremental rendering approach for ACTIVEMATH. Instead of sending all content through the pipeline before displaying the final result to the user, the presentation pipeline is driven from the view layer, i.e. from the VELOCITY template. The controller logic only collects the IDs of the content items to display on a page, and – along with all other request-dependent data – passes it on to the appropriate VELOCITY template. This template has access to a special helper bean which provides a high-level function to trigger the presentation pipeline for a single content item. The helper bean will take care of fetching an item, transforming it and rendering it directly to the request's output stream. This approach minimizes the time until the user sees the start of a new page, along with the first content element. While still rendering the content below, the user can already start reading the top of the page, which makes page rendering appear much faster, even if the server is not generating the page faster than it did before.

2.5 Caching

Caching potentially improves the performance as expensive fetching, transformation or other processes are only com-

puted once. Figure 2 indicates three points in the presentation pipeline at which caching can take place. First, caching untransformed content after *fetching* it from the knowledge bases makes sense if the databases can be distributed anywhere in the web and for those modules (such as a glossary) that use only the content meta information, such as the title of an item.

Secondly, items can be cached after the *transformation*. As applying XSLT-transformations is very expensive and requires a lot of resources, this cache potentially yields the best improvements. The cache key is the triple *(ID, format, language)*.

A third option is to cache the results of the *compilation*, for instance the PDF version of a large page or book. Here, the generated file can be stored in a file-based cache on the server to avoid the expensive re-compilation operation.

2.6 Redesigning the User-Interface

User-interface elements (e.g., menu bars) are represented in the view layer. As a result, changing the appearance of the user-interface only requires changing the VELOCITY templates. They are written in a very easy to learn syntax and contain mostly presentation code (HTML). Hence, adapting ACTIVEMATH's current university level interface to suit the requirements of, for instance, a school is almost the work of an HTML designer; and migrating the HTML code from prototypes to the templates is a small task.

Moreover, even altering the pedagogical approach underlying the learning environment is relatively easy. We recently adapted ACTIVEMATH to follow a constructivist learning approach. The current central metaphor of ACTIVEMATH is that of a book in which a learner navigates. In contrast, the constructivist approach sees the learner as being motivated by daily life problems, searching on his own for the basic knowledge required to solve the problems. Despite the pedagogical differences, the presentation architecture can be re-used entirely. It was an easy task to adapt the presentation architecture and within four person-months, the design (mostly from HTML prototypes) and functionalities were ready. Figure 3 shows a screenshot of the adapted user-interface.

3 Evaluation

3.1 Estimation of performance gain

By comparing the costs of generating output in the new and old architecture, we want to get a rough estimation of the performance gain due to the caching. Therefore, costs units are assigned to the different stages of the output generation: The learner's request to the web server costs one unit. Since only HTML output is considered here, the compilation stage has no cost. The personalization stage amounts to 5 units. The following stages assembly, transformation, and preprocessing (including fetching) are assigned to 2, 30, and 10 units, respectively. Obviously, the transformation stage has the highest cost.

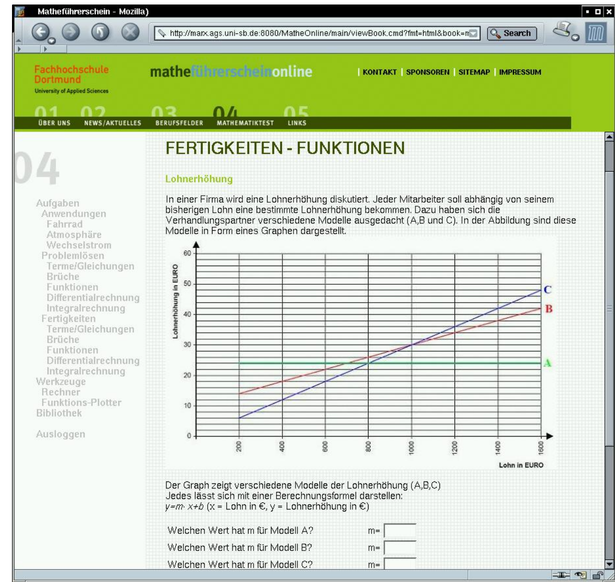

Figure 3. A different user-interface.

In the old architecture without caching, each stage has to be traversed to generate a page, i.e. for each page the cost of its generation amounts to 48 units $(1 + 5 + 2 + 30 + 10)$. However, in the new architecture with caching, in average only 3 of 10 user requests require personalization and assembly (i.e. 7 of 10 user requests are referred to the cache), and only 3 of 100 requests need extra transformation and preprocessing. As we have seen above, the transformation stage is especially costly, followed by the preprocessing stage. Therefore, a considerable reduction of cost can be expected from the new architecture, and indeed, the average cost for a page request sums up to 4.3 units $(1 + 0.3 * (5 + 2) + 0.03 * (30 + 10))$. This means that the cost reduction is about one tenth! In the following section we provide evidence for this theoretical analysis from experiments with simulated page requests.

3.2 Simulation Results

In order to get real-use data, we developed a stress tool for the ACTIVEMATH web-server. This tool simulates concurrent access to the server by automatically generating page requests. An adjustable number of virtual users "browses" through ACTIVEMATH by following menus or navigating back and forth through the content. The number of requests and the maximum time span between two request is configurable. The stress tool collects and calculates the average connection, throughputs, and latency rates.

Figure 4 contains the latency rates (the averaged value of seconds that passed until a request was answered by the server) of 4, 8, 16, and 32 users, with each user following a random sequence of 20 requests. ACTIVEMATH and the

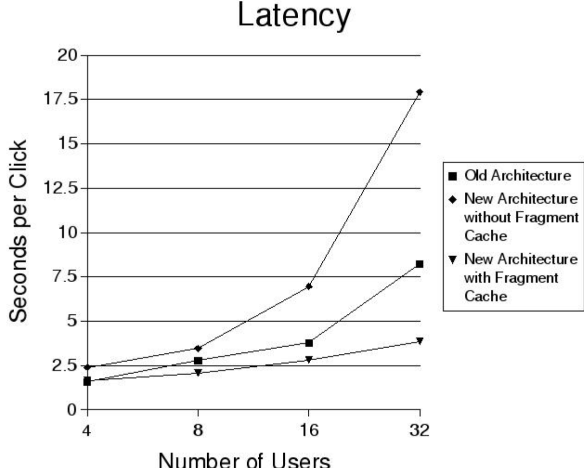

Figure 4. The averaged latency rates.

knowledge base run on the same machine to minimize the effect of network delays. The tests were performed on a 900 Mhz AMD Athlon CPU, 512 MB RAM, running Linux. We chose a standard work-place computer instead of a high-tech server in order to get a realistic idea of the different performances. Using a server, the results may have been less obvious.

Parts of the results were quite suprising. For instance, we did not expect the old architecture (OA) to outperform the new architecture without chaching (NA-C). One possibile cause may be that the XSLT stylesheets were not optimized in view of the new architecture; actually both used the same stylesheets. As a more pleasent suprise, the data very clearly shows the effect of fragment caching. The response times of NA-C become unacceptable as soon as more than 16 users are accessing the server simultaneously. With fragment caching (NA+C), the latency grows much slower. Although the number of users raises from 4 to 32, the response time stays below 4 seconds. Not accounted for in this figure are the effects of the incremental rendering. Almost instantly after having sent a request, the user will see the first item of a page in his browser, a substantial improvement of the perceived performance.

Compared to the performance of sites such as Google, these results may seem somewhat ludicrous. However, one should keep in mind the technical side, i.e., the limited power of our test server, and even more importantly, the setting for which ACTIVEMATH was designed. ACTIVEMATH is to provide individual learning support in the context of lectures and classes. In these cases, the number of users is limited to at most a hundred, and not all users will access the system simultaneously. Therefore, the new presentation architecture of ACTIVEMATH seems to be reasonably efficient for these situations.

4 Related Work and Conclusion

A large number of systems in education offers individualization of content (for a compilation see [1]). Unfortunately, we do not know of any publication that provides a technical description on a level of detail as provided here.

Cocoon (`http://cocoon.apache.org/`) is a web development framework, which is based heavily on XML and XML transformation pipelines. In contrast to the Cocoon developers, we believe that XML documents are not an ideal basis for application objects. For example, in ACTIVEMATH it is much more efficient to represent a content item in a POJO (plain old Java object) than holding this information in a JDOM XML object (Java Document Object Model). In our case, the XML object increases memory requirements by a factor of 5 to 10. In addition, an XML object typically only provides a tree view of an object, which is not always appropriate for the application, and forces the application to cope with an external storage structure. Besides, a POJO easily provides type-safe access to data elements, which is not available for XML objects and a DOM based API. Therefore, we try to restrict usage of XML to the edges of our application (data and output) wherever possible.

To summarize, we propose a layered presentation architecture that efficiently realizes AH-techniques and can be applied in all settings where XML-items serve as the basis of personalization. With our 2-stage, hybrid XSLT and template engine approach, we obtain the following benefits: Firstly, XSLT-transformed items are independent of user data. This allows for effective caching, since content is personalized only late in the presentation pipeline. Secondly, XSLT stylesheets are not dependent of the view layout. Instead, they focus on transforming content into different formats. Thirdly, the view layout (e.g. in HTML) is only contained in template files, where it is easily changeable even by many users including non-programmers. Even more, the whole look&feel of our application can be exchanged by just altering the set of templates.

References

[1] P. Brusilovsky and M. T. Maybury. From adaptive hypermedia to the adaptive web. *Communications of the ACM*, 45(5):30–33, 2002.

[2] Michael Kohlhase. OMDoc: Towards an internet standard for mathematical knowledge. In Eugenio Roanes Lozano, editor, *Proceedings of Artificial Intelligence and Symbolic Computation, AISC'2000*, LNAI. Springer Verlag, 2001.

[3] E. Melis, E. Andrès, J. Büdenbender, A. Frischauf, G. Goguadze, P. Libbrecht, M. Pollet, and C. Ullrich. ACTIVEMATH: A generic and adaptive web-based learning environment. *International Journal of Artificial Intelligence in Education*, 12(4):385–407, 2001.