A Model-Driven Runtime Environment for Web Applications

Jörg Pleumann and Stefan Haustein

Computer Science Dept. VIII/X
University of Dortmund
Germany
{joerg.pleumann,stefan.haustein}@udo.edu

Abstract. A large part of software development these days deals with building so-called Web applications. Many of these applications are database-powered and exhibit a page layout and navigational structure that is close to the class structure of the entities being managed by the system. Also, there is often only limited application-specific business logic. This makes the usual three-tier architectural approach unappealing, because it results in a lot of unnecessary overhead. One possible solution to this problem is the use of model-driven architecture (MDA). A simple platform-independent domain model describing only the entity structure of interest could be transformed into a platform-specific model that incorporates a persistence mechanism and a user interface. Yet, this raises a number of additional problems caused by the one-way, multi-transformational nature of the MDA process. To cope with these problems, the authors propose the notion of a model-driven runtime (MDR) environment that is able to execute a platform-independent model for a specific purpose instead of transforming it. The paper explains the concepts of an MDR that interprets OCL-annotated class diagrams and state machines to realize Web applications. It shows the authors' implementation of the approach, the Infolayer system, which is already used by a number of applications. Experiences from these applications are described, and the approach is compared to others.

1 Introduction

A large part of software development these days deals with building so-called Web applications, that is, server-sided applications that are remotely accessed via the Internet using a standard Web client like Netscape or the Internet Explorer. Communication between client and server is based in the Hypertext Transfer Protocol (HTTP) and uses the Hypertext Markup Language (HTML) for content description. For nontrivial applications, static HTML pages are usually not sufficient – instead, each page exists in two variants: The server holds the original page that consists of HTML code and embedded scripting commands which, for example, access the content of some underlying database. This template-like page is processed on the server, resulting in a pure HTML page which is then delivered to the client.

Web applications often employ a three-tier architectural approach: The lower tier provides a persistence mechanism for the entities the application deals with. The upper tier provides either the HTML user interface meant to be consumed by humans or a communication interface for other applications based on, for example, the Simple Object Access Protocol (SOAP). The middle tier ties the other two together and implements the application's business logic. While this approach is relatively common, it raises a number of problems:

– The database used in the persistence tier is likely to be a relational one. Given that the rest of the application is modeled using the Unified Modeling Language (UML) [1, 2] and later implemented using an object-oriented programming language like Java, a mapping between the object-oriented and relational worlds is necessary. This mapping is further complicated by the need for a normalization of database tables and the expressive mismatch between the Standard Query Language (SQL) and a modern object-oriented language like Java.
– As mentioned, most approaches use some form of scripting language to separate static and dynamic portions of a Web page. While it is possible that the scripts are implemented in (roughly) the same language as the rest of the application – like in the combination of Java and Java Server Pages (JSP) – , this is not a necessity. It may well be a different language like PHP or Perl, which results in at least five languages being used in the overall system: UML, SQL, HTML, Java plus the scripting language. This poses high demands on the developers' skills, it raises development time and cost and it complicates maintenance.
– In a significant number of cases, the application's business logic is pretty uniform. Take, for example, the typical simple Web application used to realize the Internet presence of a university department (see Fig. 1). The database stores instances of some entity classes, and the user interface provides access to them. Often, even the navigational structure of the user interface is close to the entities' class structure, that is, there is a correspondence between domain classes and HTML pages used to display, manipulate or query instances of these classes. If the logic is not application-specific, it seems unnecessary to explicitly model and implement it. Instead of going through the effort of the full three-tier approach, one would want to focus on the entities and their presentation in the user interface and leave the rest to a tool.

A solution to these problems, as mandated by OMG, is the use of Model Driven Architecture (MDA) [3, 4]. For a given problem, MDA proposes that first a platform-independent model (PIM) be created. This model is then transformed to one or more platform-specific models (PSM) using appropriate transformation rules. In the given domain of Web applications, the PIM could encompass application-specific information like the domain model and the application's business logic. From the potentially infinite number of possible PSMs, one could incorporate a user interface based on HTML and a persistence layer empoying a relational database. Theoretically, a PSM can be refined into an even more

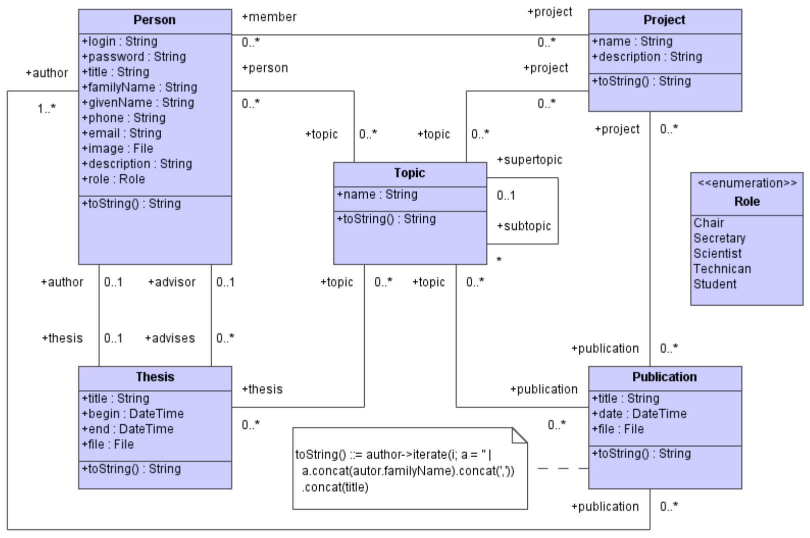


Fig. 1. A (simplified) domain model of a university department

platform-specific model. Yet, at some point programming language code has to be emitted that can be compiled into an executable application.

From a programmer's point of view, MDA is not completely new. It is extending the traditional idea of a compiler to the earlier phases of the software development process, that is, to the models. While this is surely a powerful idea, it has some consequences for the overall process as well as the application under development:

– Evolution is complicated. As Heckel and Lohmann [5] have noted before, MDA doesn't pay enough attention to functional evolution of the system. Every such evolutional step, for example a new requirement, induces changes to the PIM or the specification of the transformations from the PIM to the PSMs. In either case, the whole transformational chain up to the executable application has to be applied over and over again. Given that a large part of Web application development deals with the creative process of designing an appropriate user interface, that is, small changes to HTML pages (or their equivalent in the model) are made and evaluated, these time-consuming transformations are likely to hamper development progress.
– Maintenance is complicated. This is due to the fact that the application has not only undergone one transformation, as in the traditional compilation approach, but possibly several ones. Tracking a problem in the running application back to its roots in either the PIM or one of the transformations requires that the equivalent to "debugging" information is available for each model that was derived from a less-specific one. Currently, there seems to be

no solution for this problem. Also, as usual for generative approaches, the generated models and source code are likely to be unreadable, since they are not meant to be consumed by humans.

– The process is one-way. While it is theoretically possible to modify models generated during a previous transformation step, this is not recommended. Manually changing a PSM or some generated source code potentially results in an inconsistent description of the whole system architecture, since these changes are neither reflected in the other models nor gained "legally" through a transformation. They are lost when a complete rebuild of the system is done starting from the PIM. Thus, until there exists a means to propagate manual changes in any model to the rest of the system architecture, it is best to treat generated models and source code as read-only.

Since all three problems stem from the multiple transformations (or compilations) inherent in the MDA approach, the authors were looking for a solution that suited the Web application domain better. As a result, we propose a slight variant of MDA that does not compile PIMs, but interprets them instead. In this approach, the transformation from the PIM to the PSM is handled implicitly by a model-driven runtime (MDR) environment. Where MDA potentially transforms object-oriented concepts to non object-oriented ones (as in the case of the relational database), an MDR implements selected parts of the UML metamodel and interprets them for a given application domain. Just like it is possible to derive multiple PSMs from a single PIM in the pure MDA approach, it is possible to have a number of very different MDRs executing the same PIM for different reasons. In our case, the MDR of interest is one that allows us to build Web applications as described above.

The rest of this paper is organized as follows: Section 2 presents the basic concepts of an MDR for Web applications. It shows how UML is used to describe static and dynamic aspects of a Web application and how these descriptions are interpreted at runtime. Section 3 presents our implementation of an MDR for Web applications, the Information Layer system, or Infolayer for short. Section 4 describes examples of concrete applications realized with the approach, with experiences from these being given in section 5. The final sections 6 and 7 compare our approach to others and draw a conclusion.

2 Concepts

The basic idea of an MDR is to not have to go through all the transformational steps from the PIM via the PSM and the generated source code to a working application. Instead, the MDR is to interpret, or *execute*, the PIM itself: Any class diagram designed in a Computer Aided Software Engineering (CASE) tool and exported to the standard Extensible Metadata Interchange (XMI) format supported by most contemporary tools should be sufficient to invoke the system. The MDR then provides a user interface and persistence mechanisms otherwise gained though transformation or explicit modeling/implementation. To achieve this, the model information, possibly annotated with constraints specified in the Object Constraint Language (OCL) [6], is interpreted in several ways:

1. The model drives the database.
2. The model drives the user interface.
3. The model provides business logic.

Fig. 2 depicts the system at a very high level of abstraction. We are going to detail each aspect in one of the next sections.

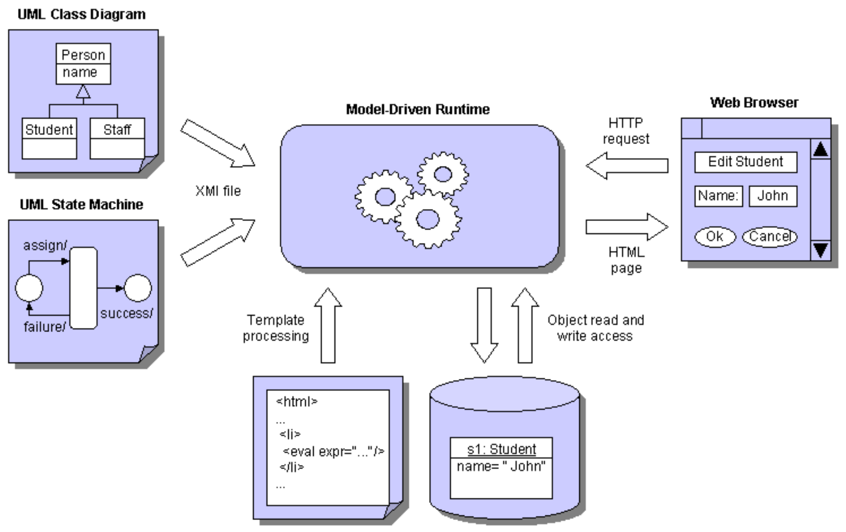


Fig. 2. Overview of an MDR-based Web application

2.1 The model drives the database

First of all, the MDR needs to provide a persistence mechanism that allows to create, access, modify, and delete instances of the classes defined in the model. Attributes can utilize the usual primitive types Boolean, Integer, Float, String, and DateTime. An additional built-in type File can provide for easily storing binary files and thus supports a limited form of content management system (CMS) functionality. Associations between classes are first-order elements that are kept consistent by the system according to the multiplicities at the association ends. No normalization of tables is necessary, as in the relational database case.

All changes are made persistent to some underlying storage, which may employ either a system-specific mechanism based on, for example, the Extensible

Markup Language (XML) or an existing database system. Changes that violate OCL constraints specified in the class diagram are denied. Practically speaking, the MDR allows to construct a persistent object diagram that conforms to the given class diagram.

Operations are supported, too, as long as they are queries (that is, side-effect free). The reason for this requirement is the language we chose to "implement" operations: OCL serves as our primary language for evaluating any kind of expressions in the system.

The system knows several predefined classes. One of them is the usual `Object` class that forms the ultimate ancestor of any user-defined class. `Object` provides a built-in operation `toString(): String` that is used to derive printable text for any object in the system. It is used and redefined in the same way as, for example, in the Java class libraries. Another predefined class `User` serves as the basis for user management, authentication and access control to classes and objects based on permissions (again expressed in OCL).

2.2 The model drives the user interface

Theoretically, an MDR can be accessed in numerous ways, but for the moment we're only interested in Web-based access. For this purpose, the system needs to run inside a servlet-capable Web server or provide Web server functionality itself. For each client accessing the system, a generic user interface can be generated on-the-fly, based both on the class and the object information:

– The interface shows a clickable inheritance tree of known classes. When a specific class is selected, its current instances are listed, and individual instances can be selected or created.
– For each instance, the system shows a list of all attributes and associations, with associations being rendered as hyperlinks to the associated objects. The latter allows the user to easily nagivate through the whole object diagram.
– When editing an object, list boxes are used where possible to restrict the user's input to sensible choices – like exactly those objects that can participate in a certain association.
– A very similar screen is used for querying the database.

Fig. 3 depicts the user interface generated for the `Thesis` class of the university example. Underlined strings denote hyperlinks. Arrows are used to clarify which parts of a class declaration influence which parts of the user interface.

Being relatively simple and thus maybe not sufficient for real-life applications, the generic user interface needs to be tailorable to specific needs. We find it easiest to achieve this tailoring by an XML/HTML-based template mechanism, since HTML is also the target language of the system and a number of powerful and mature tools exists for creating HTML pages. We assume that people responsible for the more artistic design of a system will prefer these over a CASE tool. The template mechanism allows to change the default output generated for a certain class or for objects of a certain class. It is mainly used to

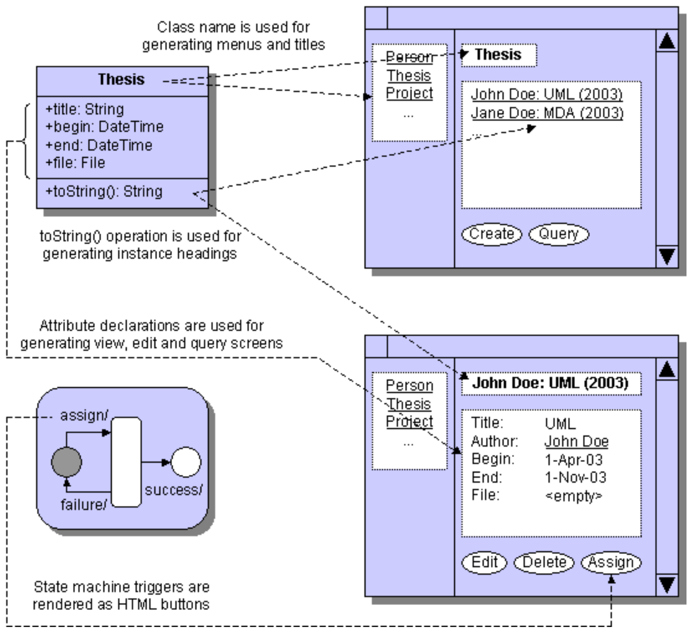


Fig. 3. Generated user interface

control the layout of pages generated by the system. Yet, it also allows to realize systems whose navigational structure is far from the assumption of one class or object being displayed at a time.

The template mechanism is somewhat similar to the mentioned approach of embedding a programming or database query language into HTML pages. Yet, it is very different in the language we propose to use: Again, OCL is used to retrieve a set of objects matching a desired criterion. A few additional XML elements can allow for a limited degree of "control flow" while generating an HTML page. As an example, there can be an element that allows to iterate over the constituents of an `OclCollection` (the result of a `<class>.allInstances.select(<condition>)` expression) and output a certain HTML fragment for each. Another XML element provides an equivalent to the usual `if-then-else` construct known from programming languages. Its condition is an OCL expression that evaluates to a boolean value. Both are depicted in Fig. 4.

The templates follow the inheritance rules dictated by the class hierarchy: Subclasses inherit the templates defined by their superclasses, but can override them. If no specific templates are defined at all, classes inherit their templates from `Object` – which results in the default output behaviour described above.

```
<h1>Theses Available</h1>

<ul>
  <!-- The inner block is repeated for each item.  -->
  <!-- 't' is the namespace for template elements. -->

  <t:iterate expr="Thesis->select(author->isEmpty())">
    <li>
      <!-- 'self' holds the current iteration item (optinal). -->
      <!-- 'eval' evaluates and prints the given expression.  -->

      <t:eval expr="self.title"/>

      <!-- 'if' encloses a conditional block. -->

      <t:if expr="advisor->notEmpty()">
        (advice by <t:eval expr="advisor.givenName"/>
                   <t:eval expr="advisor.familyName"/>)
      </t:if>
    </li>
  </t:iterate>
</ul>
```

Fig. 4. Template incorporating OCL expressions

2.3 The model provides business logic

Experience with Web applications shows that a number of systems have more or less workflow-like characteristics. Take, for example, the university department model from Fig. 1. Such an Internet presence would usually provide a list of master theses, each of which can be in different states: A thesis can be available, it can be reserved for a student who is writing a proposal, it can be work in progress, and it can be finished. The possible transitions between these states are restricted, and the user interface should enforce these restrictions. For example, one should be able to go from ”in progress” to ”finished”, but not back. Such behavior is easily specified as a UML state machine (see Fig. 5).

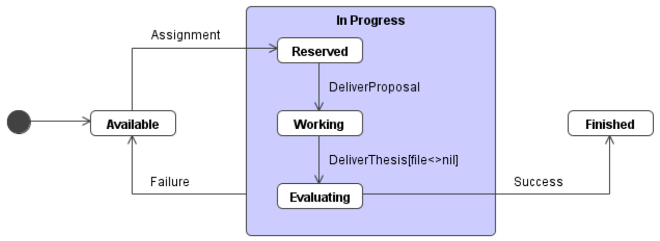


Fig. 5. State machine for the `Thesis` class

The MDR is able to interpret state machines as additional business logic of the system. Every class in the domain model can be annotated with a state machine that describes its behaviour. Once a new object is created at runtime, it not only has all its attributes set to default values, but also starts in its initial state(s).

When the object is displayed, the HTML user interface shows a list of buttons representing the potential triggers. These are the triggers of exactly those transitions that are enabled, or, more precisely, would be enabled if these events were to enter the system (see Fig. 3). The buttons can take into account any guard expressions attached to the transitions. These guards are, again, specified in OCL and may encompass all objects in the system. If a button is pressed, the enabled transitions are taken, resulting in a new active state configuration. Is also makes sense to allow special actions here that are able to modify an object's attributes by some form of assignment. The new state configuration is made persistent with the object, just as the attributes and associations are. Once a state machine reaches a terminal state, the object it belongs to is deleted.

If required inside a template, the current active state configuration of an object can be queried using the `OclInState(<state>)` function, and HTML output can be created accordingly. As an example, one could display a longer textual description of the current state to provide the user with some assistance on the current position in terms of the workflow and what actions are possible.

3 Implementation

We have implemented the above ideas in our Infolayer system. The roots of this Java-based system go back to the COMRIS [7] project funded by the European Community. Fig. 6 shows a very rough approximation to the architecture. At the heart of the system lies an implementation of selected portions of the UML metamodel:

– A core part, implementing basic properties of all model elements. This part has a loose conceptual relationship to the UML meta-metamodel, but it is not an implementation of the Meta Object Facility (MOF).
– A part that implements UML classes and objects, including all the necessary properties like attributes, associations, methods, inheritance, etc.
– A part that implements UML state machines, including a runtime component that is able to simulate a state machine.
– An OCL parser and evaluator. This component implements the OCL language as defined in the UML 1.4 specification. As mentioned, OCL is used to specify (side-effect free) query operations. For pragmatic reasons, we also allow some additional (non-query) constructs. These encompass object creation and destruction as well as value assignment and can be seen as a limited subset of Action Semantics.
– An XMI loader that allows to feed the model into the system. Since XMI operates on the UML meta-metamodel, this part requires the core part only and is able to handle arbitrary metamodel elements by means of the reflective capabilities introduced in the core model element.

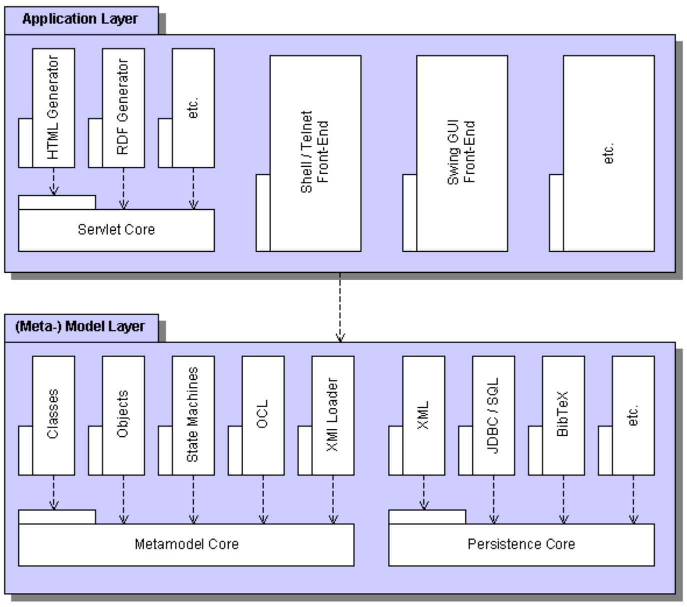


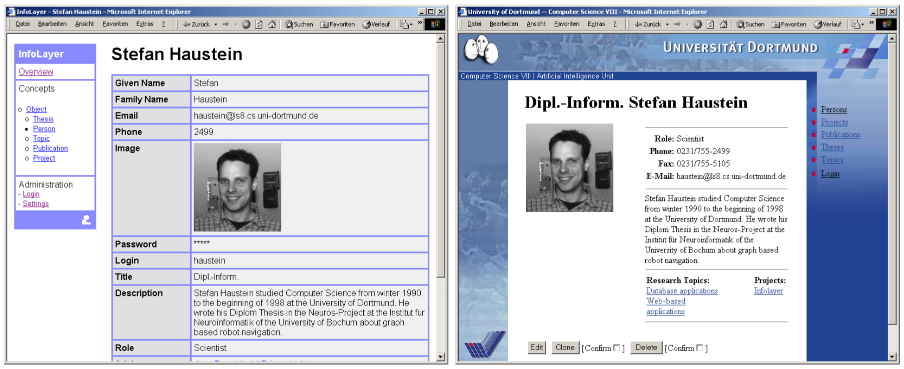
Fig. 6. Architecture of the Infolayer

– A part that implements an XML-based persistence mechanism as well as other ones, for example one based on Java Database Connectivity (JDBC) that allows the Infolayer to integrate existing relational database. Actually, it is even possible to have the Infolayer operate on a BibTeX file.

All these components contribute to the system's model layer or back-end, which is still largely application-independent. Atop the model layer lie different application front-ends which provide the functional equivalent to transformations from the PIM to a PSM in MDA. One of them is the servlet already mentioned in the previous sections. The servlet uses template mechanisms to generate output based on the model information. HTML is one possible output format, but it's not the only one. Attempts have also been made to produce output conforming to the Resource Description Framework (RDF) to open the system for the Semantic Web [8]. Further template sets could be used to address mobile phones using the Wireless Markup Language (WML) or a limited subset of HTML.

Fig. 7 shows the HTML user interface for the university example described above, both with and without using templates.

In addition the the servlet, several other front-ends exist. A command-line client can be used to access the system using OCL only, and a similar client uses the Telnet protocol to access an Infolayer installation from a remote host. A Swing-based graphical client that adapts its user interface to the model has also been implemented, but is currently not maintained, because the focus lies on


InfoLayer - Stefan Haustein - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Verlauf
InfoLayer
Overview
Concepts
Object
Thesis
Person
Topic
Publication
Project
Administration
- Login
- Settings
University of Dortmund -- Computer Science VIII - Microsoft Internet Explorer
Universität Dortmund
Computer Science VIII | Artificial Intelligence Unit
Persons
Projects
Publications
Theses
Topics
Login
Edit
Clone
[Confirm ☐]
Delete
[Confirm ☐]

Stefan Haustein

Given Name	Stefan
Family Name	Haustein
Email	haustein@ls8.cs.uni-dortmund.de
Phone	2499
Image	
Password	*****
Login	haustein
Title	Dipl.-Inform.
Description	Stefan Haustein studied Computer Science from winter 1990 to the beginning of 1998 at the University of Dortmund. He wrote his Diplom Thesis in the Neuros-Project at the Institut für Neuroinformatik of the University of Bochum about graph based robot navigation.
Role	Scientist

Dipl.-Inform. Stefan Haustein

Role: Scientist
Phone: 0231/755-2499
Fax: 0231/755-5105
E-Mail: haustein@ls8.cs.uni-dortmund.de

Stefan Haustein studied Computer Science from winter 1990 to the beginning of 1998 at the University of Dortmund. He wrote his Diplom Thesis in the Neuros-Project at the Institut für Neuroinformatik of the University of Bochum about graph based robot navigation.

Research Topics: Database applications, Web-based applications
Projects: Infolayer

Fig. 7. User interface with and without templates

the web client. Finally, a client implementing the Simple Object Access Protocol (SOAP) allows to remotely access the Infolayer from third-party applications.

4 Applications

In spirit of "eating one's own food", that is, the idea that a (software) engineer should always be the first one to apply his own systems, the Infolayer is being actively used in a number of different projects. The largest of these applications is probably the Web presence for MuSofT (`http://www.musoft.org`), a distributed project that develops multimedia teaching material for software engineering education in Germany [9]. The goal of this web presence is to manage and distribute the material contributed by the various project partners and to facilitate sustainable (re-) use amongst its users. The corresponding database is rather complex: It not only features authors, their (binary) material and access rights, but also metadata conforming to the Learning Objects Metadata (LOM) [10] standard as well as a subset of the ACM computing classification system [11], both of which allow for proper structuring and efficient retrieval inside the database.

A second application is the Java 2 Micro Edition (J2ME) Device Database (`http:// kobjects.org/devicedb`), a database of mobile phones and personal digital assistants (PDAs) supporting J2ME. Although these devices follow a common standard, they all have their own bugs, peculiarities and limitations – which is critical information from the point of view of a developer who, naturally, can't own all the devices available in the world. The database accumulates this information. It receives "live" data from a small benchmarking application that is publicly available and can be run on the different devices by their owners, sending its results to the Infolayer database after it has been executed.

Other applications include the Machine Learning Net (MLnet) teaching server (`http: //kiew.cs.uni-dortmund.de:8001`), a system that provides information for the artificial intelligence community, and several chairs of the University and the University of Applied Sciences in Dortmund who use it to manage their web presences. In addition to these, the Infolayer is used in numerous smaller projects where a database with web-based front-end and an easy-to-use navigational structure is required without spending much effort on its implementation.

5 Lessions learned

Some of the above applications have been in use for about two years. Experiences with these applications and with the overall Infolayer approach have shown a number of things. For this paper, we'd like to concentrate on the areas of feasibility, methodology and OCL usage.

5.1 Feasibility

First, the general approach of modeling relevant structural and dynamic parts of a Web application in a CASE tool and then executing this model works well. For the applications mentioned in the previous section, there was practically no additional Java programming necessary (only the MuSofT application required one additional class to handle e-mail notifications sent to interested users once a learning object changes). With the model itself becoming executable, we were able to produce a working prototype for any of the applications very early. If a database schema proved insufficient for the application, it was possible to go back to the CASE tool, change the model, and then execute it again. No implementation effort was spend on thrown-away prototypes, which makes the Infolayer ideal for a rapid application development (RAD) approach in the Web application or database context. We think a similar approach would be possible in other areas, too.

5.2 Methodology

Second, a methodology or "best practice" for working with the Infolayer has evolved over time. It emcompasses these steps:

1. A domain model consisting of OCL-annotated UML class diagrams and possibly UML state machines is designed using one or more iterations of designing and testing a prototype, as described above.
2. A first attempt is made to tailor the system's layout (to personal taste or a given corporate design) using one very simple template that provides a basic frame and a main navigation structure. This is usually the point at which the system can actually be be utilized by its users, that is, the Web application's database can be filled with content.
3. The page layout for individual classes can successively be improved, or the system's whole navigational structure can be changed according to the specific needs dictated by the application.

4. The model itself can be modified, too, as long as these changes only introduce elements (classes, attributes, associations, constraints) into the system that are consistent with existing instances. We hope to loosen this restriction using refactoring facilities in the near future.

5.3 OCL usage

Third, the decision to use OCL as a query language inside the system proved to be very helpful. A previous version of the Infolayer used the Object Query Language (OQL) [12] instead, so the authors are able to draw a comparison here: OQL is basically a slight syntactic adaption of SQL to the object-oriented world. Queries that encompass multiple associations with a cardinality greater than one tend to become lengthy and unreadable, because they result in nested `select` statements. The implicit `collect()` in OCL expressions allows much shorter and more intuitive queries.

As a result, we propose to use OCL – possibly with some syntactic extensions for database updates – as an general access language for OODBMS. An alternative would be the use of a subset of UML Action Semantics, probably with a surface language that resembles OCL.

6 Related work

The Infolayer is not the first system that is based on the idea of interpreting and executing a graphically specified formal model. Harel's Statemate tool [13] uses state machines to describe and simulate system behavior. The Real-Time Object-Oriented Modeling (ROOM) [14] language specifies both system structure and behavior using a mixture of classes and state machines. Both systems are targeted at generating code for embedded systems, not for database systems or Web applications. As a result, user interface considerarions are not a concern.

Horrocks [15] uses state machines for modeling user interfaces, where we are using them for modeling an object's behavior and have a user interface largely based on the domain model. If we were to allow additional graphical specification of user interface structures, we would likely use activity diagrams instead of state machines to support the description of "wizard-like" dialog sequences that implement certain use cases.

Riehle [16], Mellor [17] and Frankel [4] mention UML virtual machines (VM) capable of interpreting arbitrary UML models. Yet, these VMs seem to be more of a means to simulate and check a given model or use it for research purposes in areas like refactoring [18]. As far as we can see, most of them are not targeted at a certain application domain, as is the Infolayer, yet the systems are very close in spirit to ours.

There's also a number of approaches trying to employ the UML for creating Web applications. Conallen [19] uses UML stereotypes to model various aspects of a Web application, from client and server components to details of individual HTML pages. While this approach pays off for very large applications that incorporate traditional executable code development, we find it overly complex for the average small or medium Web application. Baumeister et. al [20] describe a systematic design method for Web applications combining ideas from the Object Oriented Hypermedia Design Method (OOHDM) [21] and UML. The system specification is divided into a conceptual model, which is roughly equal to our domain model, and a navigational model. Since our premise is that the system's navigational structure is close or equal to the entity structure anyway, we don't see the necessity for the navigational model, and the examples given in [20] seem to concede more to our point than to theirs.

WebML [22] is a high-level specification language for data-intensive Web applications. It seems closest to our approach in that it focuses on an entity-relationship model, that is, on a subset of UML specifying the data classes of interest. One then composes Web pages from these entities and high-level components like buttons, indexes etc.

Interestingly, all three approaches try to model HTML page layout by means of UML and find equivalents to single HTML elements using stereotypes. In our opinion this is putting the cart before the horse. The purpose of a Web application is determined by its content, that is, the entities it deals with, so these should come first and be central to the whole development process. Also, HTML is just one of many possible user interfaces for accessing the application. For HTML-centric approaches, incorporating alternative user interfaces or handling changes to the entity structure will be complicated. For model-based approaches – be it pure MDA or our MDR variant – this is easily done by either adding a new transformation or implementing a variant of the runtime environment.

7 Conclusion and Outlook

We have presented a novel approach to developing Web applications. It is highly model-driven, but instead of transforming a PIM to a PSM as in the MDA case, it directly interprets or *executes* the domain model. For that purpose we have the notion of a model-driven runtime environment that accepts UML models consisting of class diagrams and state machines and makes this model accessible via a servlet. A default HTML user interface is generated on-the-fly, but can be tailored to specific needs using a template mechanism. The template mechanism uses OCL plus a few additional constructs to access the domain model and the existing objects and renders them to HTML.

The idea has been implemented in the Infolayer system, which provides the basis for a number of different Web applications already in use. Experiences with developing and using these applications have been very positive so far, and we feel that it should be feasible to apply the same ideas to other application domains, too.

Amongst the things we consider for future extensions to the system are additional UML diagram types, refactoring facilities and incorporation of full Action Semantics.

References

1. Object Management Group: Unified Modeling Language (UML) 1.5 Specification. http://www.omg.org/cgi-bin/doc?formal/03-03-01 (2003)
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley Longman (1999)
3. Object Management Group: Model Driven Architecture (MDA). http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01 (2001)
4. Frankel, D.S.: Model Driven Architecture – Applying MDA to Enterprise Computing. OMG Press (2003)
5. Heckel, R., Lohmann, M.: Model-based Development of Web Applications Using Graphical Reaction Rules. In Pezzè, M., ed.: Fundamental Approaches to Software Engineering, Springer (2003) 170–183
6. Warmer, J., Kleppe, A.G.: The Object Constraint Language: Precise Modeling with UML. Addison Wesley (1999)
7. Haustein, S.: Information environments for software agents. In Burgard, W., Christaller, T., Cremers, A.B., eds.: KI-99: Advances in Artificial Intelligence. Volume 1701 of LNAI., Springer Verlag (1999) 295–298
8. Haustein, S., Pleumann, J.: Is Participation in the Semantic Web Too Difficult? In Horrocks, I., Hendler, J., eds.: First International Semantic Web Conference. Volume 2342 of LNCS., Springer (2002) 448–453
9. Doberkat, E.E., Engels, G.: MuSofT – Multimedia in der SoftwareTechnik. Informatik Forschung und Entwicklung **17** (2002) 41–44
10. IEEE Learning Technology Standards Committee: Final Draft of the IEEE Standard for Learning Objects and Metadata. http://ltsc.ieee.org/wg12 (2002)
11. Association for Computing Machinery: ACM Computing Classification System. http://www.acm.org/class (1998)
12. Cattell, R.G.G., Barry, D.K.: The Object Data Standard ODMG 3.0. Morgan Kaufmann (2000)
13. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology **5** (1996) 293–333
14. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley and Sons (1994)
15. Horrocks, I.: Constructing the User Interface with Statecharts. Addison Wesley (1999)
16. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The Architecture of a UML Virtual Machine. In: 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01), ACM Press (2001) 327–341
17. Mellor, S.J., Balcer, M.: Executable UML – A Foundation for Model-Driven Architecture. Addison Wesley Longman (2002)
18. Ho, W.M., Jézéquel, J.M., Guennec, A.L., Pennaneac'h, F.: UMLAUT – An Extensible UML Transformation Framework (1999) http://www.w3.org/TR/2002/CR-soap12-part2-20021219/.
19. Conallen, J.: Building Web Applications with UML. Addison Wesley Longman (2000)
20. Baumeister, H., Koch, N., Mandel, L.: Towards a UML Extension for Hypermedia Design. In: Proceedings of UML'99. (1999)
21. Schwabe, D., Rossi, G., Barbosa, S.D.J.: Systematic Hypermedia Application Design with OOHDM. In: UK Conference on Hypertext. (1996) 116–128
22. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. Computer Networks **33** (2000) 137–157