Rapid Prototyping of Web Applications combining Domain Specific Languages and Model Driven Design

Demetrius Arraes Nunes
Departamento de Informática, PUC-Rio
Rua M. de S. Vicente, 222
Rio de Janeiro, RJ 22453-900, Brazil
+55 21 2521 2848
dema@tecgraf. puc-rio.br

Daniel Schwabe
Departamento de Informática, PUC-Rio
Rua M. de S. Vicente, 222
Rio de Janeiro, RJ 22453-900, Brazil
+55 21 3114 1500 x4356
dschwabe@inf. puc-rio.br

ABSTRACT
There have been several authoring methods proposed in the literature that are model based, essentially following the Model Driven Design philosophy. While useful, such methods need an effective way to allow the application designer to somehow synthesize the actual running application from the specification. In this paper, we describe HyperDE, an environment that combines Model Driven Design and Domain Specific Languages to enable rapid prototyping of Web applications.

Categories and Subject Descriptors
H.5.4 [**Hyperext/Hypermedia**]: Architecture; Navigation. D.2.1 [**Software Engineering**]: Requirements/Specification. D.2.2 [**Software Engineering**]: Design Tools and Techniques.

General Terms
Design, Languages.

Keywords
Model-based Designs, Hypermedia Authoring.

1. INTRODUCTION
There have been several methods for Web application design proposed in the literature, such as OOHDM [15], SHDM [12], WebML [4], OOWS [13], Hera [[20], UWE [10]. Remarkably, they all follow the principles of Model Driven Design (MDD) [18]. Simply stated, this approach uses the notion of models to help the designer perform the design activity.

A model here can be seen as a simplified, textual or graphical description of the artifact being designed. Preferably, a model should have precise, non-ambiguous semantics that enables understanding of the artifact being modeled. Being a simplification of the artifact, models introduce abstraction levels. Such abstractions, if well chosen, help the designer deal with the complexity of the artifact by hiding irrelevant detail for a particular aspect or set of aspects being addressed.

Software development, according to MDD, is a process whereby a high-level abstract model is successively translated into increasingly more detailed models, in such a way that eventually one of the models can be directly executed by some platform. The model that is directly executed by a platform which satisfies all the requirements, including the non-functional ones, is also called "code", and is usually the last model in the refinement chain.

Although this approach has been used for a number of years, its adoption is not completely widespread, at least not in its pure form. A major stumbling block has been the problem that the mapping between models, especially into actually executing code, has had little or no support from tools. Therefore, designers may use the models mostly as thinking tools, and at some stage they are forced to manually map these models into code. This process is error prone, and once the code has been generated, changes or updates to the application are directly implemented in the code, instead of adjusting the models and re-generating the code.

On the other hand, several more recent proposals attempt to alleviate this problem by having automated translations (or transformations) between models, supported by appropriate tools. Among the most prominent are MDA [5] and Software Factories [6].

Specifically for designing Web applications, most of the aforementioned methods have associated development environments that support code generation from model specifications, either fully or partially automated. The code generated contains both boiler plate code that encodes the pre-defined model semantics, and user specified code that is specific to the application at hand.

There are usually two places where user-specified code appears in applications generated using such support environments – the code implementing the business logic that is specific to the application, and code in the templates that render the interfaces to the application, which typically have to access or process data values to be exhibited in the model representation, or to store values in it.

However, this code has only indirect access to the model, through implementation structures in which it is encoded. As a consequence, the designer/developer must, to some extent, understand the implementation architecture of the support environment to correctly produce the additional required code.

In this paper, we show how the HyperDE[1] environment supports the rapid prototyping of Web applications through a combination the Model Driven Development approach with the use of Domain Specific Languages (DSL's) [19]. This combination allows the designer/developer to write code by directly manipulating the models that specify the application. In addition, since the model is specified following the meta-model for a method, it also possible to dynamically manipulate the model itself during execution, which enables very concise and general applications. Consequently, scripts in the generated DSL work as very high level procedural specifications of the application.

The next section presents a summary of SHDM, the design method supported by HyperDE. Then, a brief description of

[1] HyperDE is freely available at http://server2.tecweb.inf.puc-rio.br:8000/projects/hyperde/trac.cgi

HyperDE is shown, followed by the presentation of the DSL generated by HyperDE. Then we discuss the advantages of this approach, and draw some conclusions.

2. A SUMMARY OF SHDM

The HyperDE environment allows the implementation of web applications designed using the SHDM method [11],[12]. SHDM is a model-driven approach to design web applications through five different steps: Requirements Gathering, Conceptual Design, Navigational Design, Abstract Interface Design and Implementation. Each phase focuses on a particular aspect and produces artifacts detailing the application to be run on the web. The typical workflow in producing these artifacts, and thus generating the final running application, is presented in Figure 1.

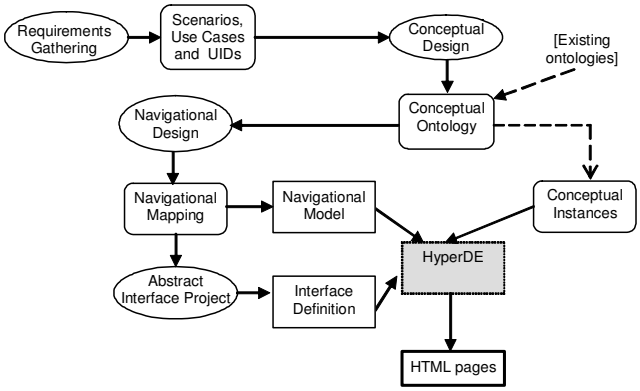

Figure 1. Simplified flow of documents in SHDM as used in HyperDE

It is a natural tendency, when building an application based on some ontology representing the problem domain, to think of it as an "ontology browser" – each "node" is essentially an "object" (i.e., a resource with associated properties), and links are certain Properties that have other such resources as values. In this sense, this ontology serves as a conceptual model of the problem domain.

One of the main insights from previous work in hypermedia application design methods, which has been kept in SHDM, is the realization that the navigation objects that are manipulated by the user are not the actual objects of the conceptual model, but *views* over these objects, defined according to user profiles and tasks to be supported by the application (see [14] for a more extensive discussion).

Accordingly, in SHDM we have taken the approach to define a Web application as a navigational view over some ontology (i.e., conceptual model) which describes the problem domain. We profit from being able to represent both data itself and its schema (meta-data) using the same formalism, since the fact that the schema can be manipulated just like any other kind of data brings greater expressiveness and conciseness to the specifications.

Since the specifications can also be treated as data, the generation of the final application is achieved by successive manipulations of these specifications, up to the point of generating the concrete interface (which must per force be in a language understood directly by the current browsers).

2.1 SHDM Meta Model

Figure 2 shows the SHDM meta model, with the main classes highlighted. The class NavClass models the navigation nodes, and the class Link models the links between them. Each NavClass has NavAttributes, NavOperations and Links, and can be a specialization of a BaseClass. Contexts are sets of objects of NavClass, defined through a query specified in one of its attributes; this query may have a parameter. Indexes are made out of IndexEntries, which contain either anchors to other indexes or anchors to elements within a context. Landmarks are anchors to either Indexes or to Context elements. Views allow exhibiting the contents of NavClass instances within some context, or exhibiting Indexes.

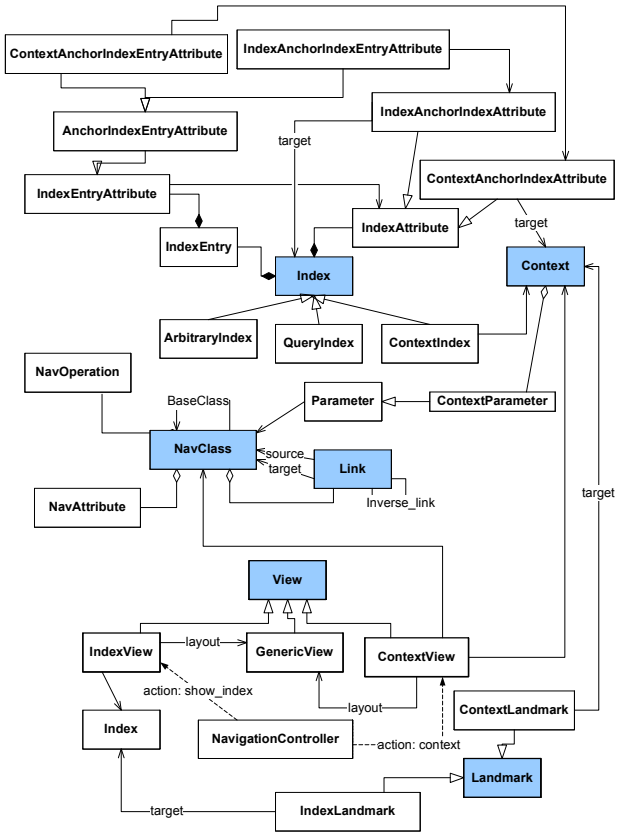

Figure 2. SHDM Meta Model

Designing a Web application using SHDM corresponds to instantiating this metamodel. The HyperDE environment supports this, and will be detailed next. However, to help explain HyperDE's architecture and the DSL language, we first give a brief sample application.

3. AN EXAMPLE USING HyperDE

Consider an academic department, where there are Professors who advise Students; both produce Publications and work in some ResearchArea. This is can be regarded as a simplified view of a departmental website whose conceptual model is represented in Figure 3, using a UML-like notation.

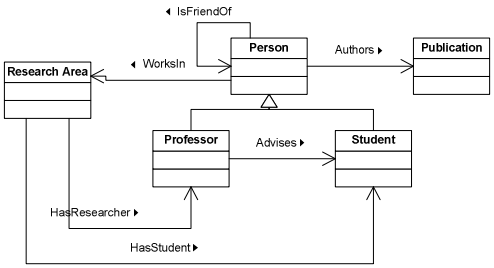

Figure 3 – The conceptual schema of an academic department website

The possible navigations are shown in the navigation contexts diagram shown in Figure 4, following the SHDM notation.

Starting with the index of Research Areas shown in Figure 5, the user chooses one, e.g., "Software Engineering", and navigates to the node exhibited in Figure 6. The index of other Research Areas on the left bar, the landmarks in the bar at the top, and the breadcrumbs are all generated automatically.

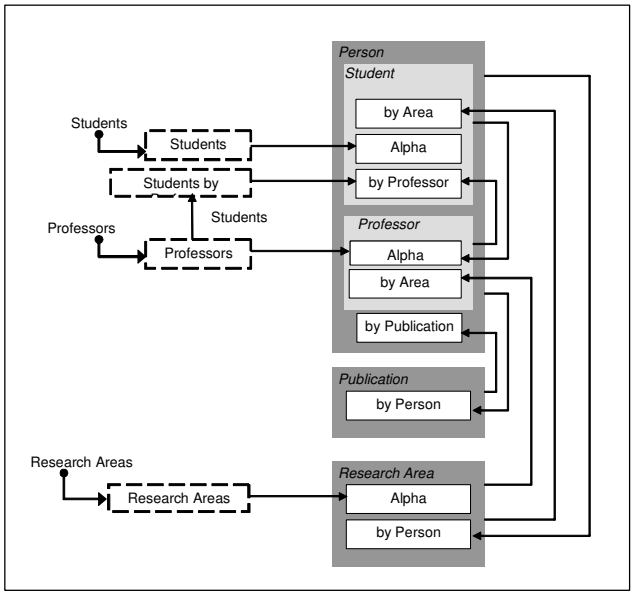

Figure 4. Navigation Context Schema for the Academic Department Website

If the user chooses a Professor in the area, say "Daniel Schwabe", he will see the screen shown in Figure 7.

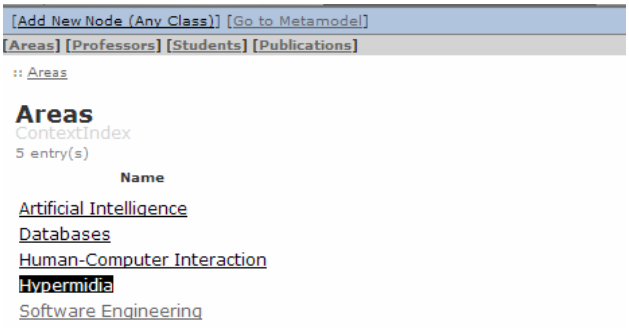

Figure 5. The Research Areas index.

In SHDM it is possible to specify that a node appears differently depending on the context in which it is being navigated. As an example, Figure 8 shows the same professor, but now being navigated in the "Professors in Alphabetical Order" context.

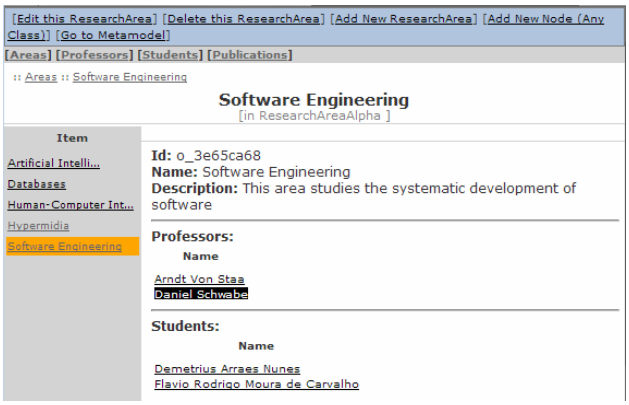

Figure 6. A Research Area.

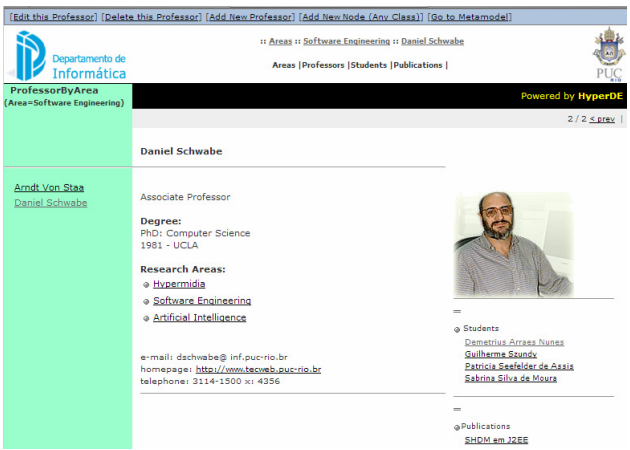

Figure 7. A Professor node in the context "Professors by Research Area".

It can be readily noticed that, besides using a different layout, additional attributes are also exhibited, such as the "Change Email" operation, and the "Students in Area" attribute.

Figure 8. The Professor node shown in Figure 7, but seen in the context "Professors in Alphabetical Order"

Whereas the layouts in Figure 7 were custom defined, the one in Figure 8 is a default layout that is applicable to all nodes, regardless of its actual class.

4. THE ARCHITECTURE OF HYPERDE

The HyperDE environment is based on the MNVC framework [9], which extends the MVC framework with navigation primitives. It allows the designer to input SHDM navigational models (the "model" in the MVC framework), and interface definitions (the "view" in the MVC framework), and generates complete applications adherent to the specification. It also provides an interface to create and edit instance data, although, strictly speaking, this should actually be part of the generated application. Figure 9 shows the architecture of HyperDE.

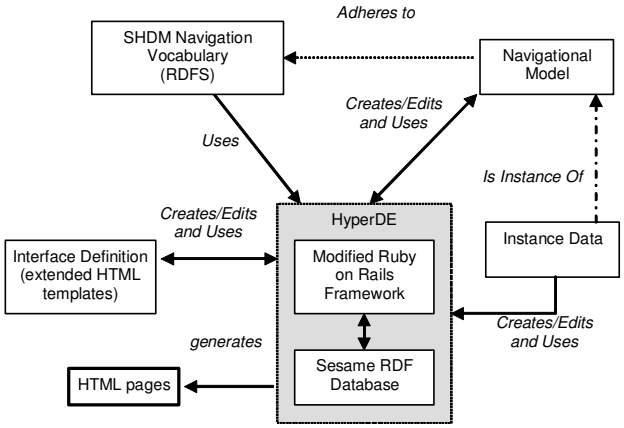

Figure 9. The architecture of the HyperDE environment.

HyperDE is implemented as a modification of the Ruby on Rails framework (http://www.rubyonrails.com), where the persistence layer (ActiveRecord) has been replaced by another one based on the Sesame RDF database. The SHDM meta-models, the user defined navigation models, as well as the application instance data, are all stored as RDF data.

All HyperDE functions can be accessed via Web interfaces. In addition, HyperDE also generates a Domain Specific Language (DSL) as an extension of Ruby, allowing direct manipulation within Ruby scripts of both the model and SHDM's meta-model.

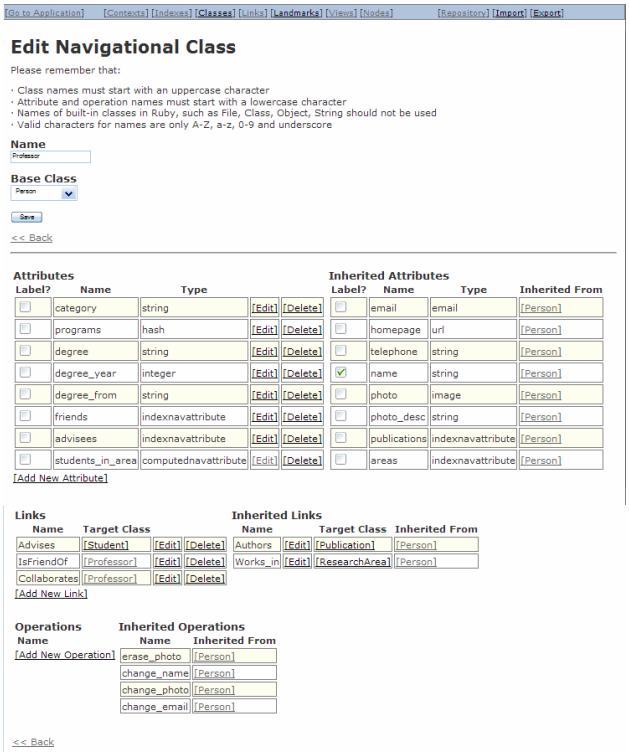

Figure 10. The interface for editing a Navigational Class

Figure 10 shows the editing interface for a Navigational Class. As expected, it allows defining all the class attributes, its superclass, its links and its operations. Notice that this screen merely instantiates the NavClass metaclass.

Figure 11 shows the interface for editing a Navigational Context. The most important attribute is the query which defines the nodes that belong this context.

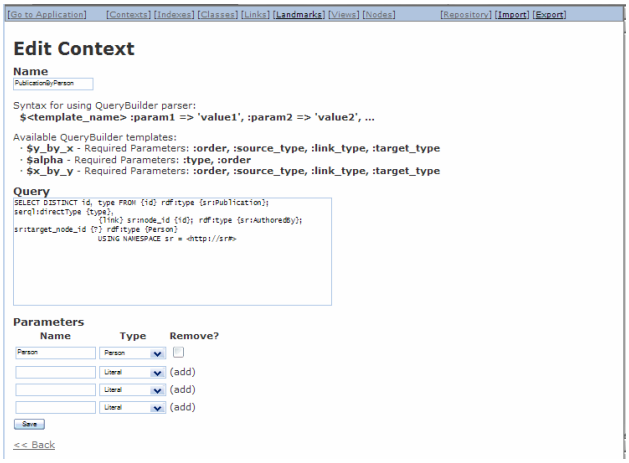

Figure 11. The interface for editing a Navigational Context.

Figure 12 shows the interface for defining an Index structure. Similarly to the Context definition, the query attribute is the most prominent in characterizing an Index.

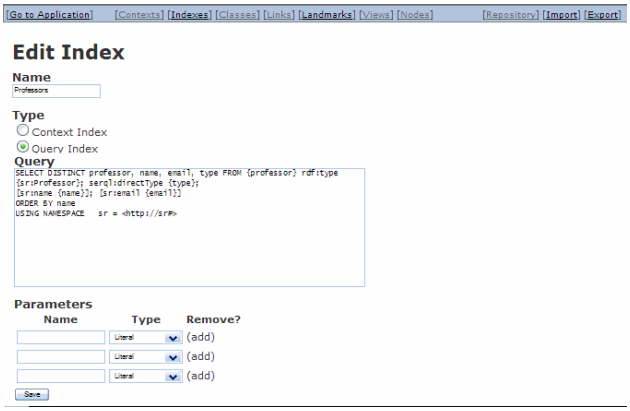

Figure 12. The interface for defining an Index structure.

In the examples shown, the queries are expressed using the SeRQL query language [3] , which allows full exploration of the underlying RDF [1] representation. The context query in Figure 11 is stated as

```
SELECT DISTINCT id, type FROM
{id} rdf:type {sr:Publication}; serql:directType
{type},
{link} sr:node_id {id}; rdf:type {sr:AuthoredBy};
sr:target_node_id {?} rdf:type {Person}
                    USING NAMESPACE sr =
<http://sr#>
```

Instead of explaining each detail, we rephrase it in natural language as "select all resources of type "Publication" that have a link of type "AuthoredBy" to a Person whose value is passed as a parameter." The parameter is indicated by the "?", which is not part of SeRQL, but is added by HyperDE.

While the use of SeRQL gives the designer the greatest expressive power, it is, in most cases, too specialized and it is not expected that the common user of HyperDE would know it. To alleviate this problem, we use a simple DSL that is based on the observation that, in practice, as observed by our experience of ten years applying this method and its predecessor, most queries fall under one of the two boiler plate formats:

1. Select X related to Y based on relation R ordered by attribute A – e.g., Publications by Person related by AuthoredBy ordered by title, or
2. Select all instances of class X ordered by attribute Y – e.g., Students in Alphabetical order of name.

Based on this observation, we allow the query above to be stated in the following DSL

1. `[x_by_y, source_type: Publication, target_type: Person, link_type: AuthoredBy, order: title]`
2. `[alpha, type: Student, order: name]`

This DSL has the advantage of making the context definition completely independent of the underlying database. In fact, we have also implemented an earlier version of HyperDE that uses a relational database as the persistence store. The disadvantage of this DSL is that it is not possible to state all possible queries in it, but, as we stated, our experience shows that the vast majority of cases are covered by these formats. Notice also that is possible to extend the DSL to other query boiler plate templates (e.g., X by Y by Z).

5. THE USE OF DSL's IN HYPERDE

We have argued that, regardless of the abstraction level of the specification language used to specify an application, there are portions which typically will require the designer/implementer to write some kind of code, such as for the business logic or for retrieving or storing values that flow in the interface.

This code can, in turn, be given in a language that is directly executable, or again resort an abstraction layer that requires further series of translations until it can be executed. In most environments, the language of choice is some programming language that is directly executable in the desired target environment.

In such cases, these programs must manipulate the model's representation in terms of the programming language primitives, which adds a layer of detail that is cumbersome at best. An alternative for this is to generate a DSL that makes the datatypes of the model also be the datatypes of some programming language. The advantages of this approach have already been argued in [6].

Consider, for example, the following piece of code, intended to manipulate an instance of a VCard, a popular ontology (see [6]).

```
DAMLModel model = … // code that loads the VCARD
ontology and some data based on that ontology

DAMLClass vcardClass =
    (DAMLClass)
model.getDAMLValue(vcardBaseURI+"#VCARD");

DAMLProperty fnProp =
    (DAMLProperty)
model.getDAMLValue(vcardBaseURI+"#FN");

DAMLProperty emailProp =
    (DAMLProperty)
model.getDAMLValue(vcardBaseURI+"#EMAIL");

Iterator i = vcardClass.getInstances();

while (i.hasNext()) {
    DAMLInstance vcard = (DAMLInstance) i.next();

    Iterator i2 =
     vcard.accessProperty(emailProp).getAll(true);

    while (i2.hasNext()) {
        DAMLInstance email = (DAMLInstance)
i2.next();

        if
(email.getProperty(RDF.value).getString().equals(
            "amanda_cartwright@example.org" ) ) {

            DAMLDataInstance fullname =
                (DAMLDataInstance)
vcard.accessProperty(fnProp).getDAMLValue();

            if ( fullname != null )
                System.out.println("Name: "+
fullname.getValue().getString());
```

```
        }
    }
}
```

The code below shows the equivalent function in a DSL defined over the ontology, on top of Ruby:

```
vCard.find_all.each { |vc|
    vc.emails.each { |email|
        print vc.fn if email ==
“amanda_cartwright@example.org” && vc.fn?
    }
}
```

Following this approach, we have defined a DSL on top of Ruby, based on the SHDM model and metamodel, in the following way

1. Each instance of NavClass becomes a Ruby class;
2. Each NavAttribute of NavClass becomes an attribute of the corresponding Ruby class. In addition, a method “find_by_xxx” is also defined, allowing to search objects of this class according to values of this attribute; (Actually, a whole family of “find_by_xxx” and “find_all_by_xxx” methods are created dynamically allowing one to write such expressions as Professor.find_by_name_and_research_area or Student.find_all_by_year_and_department)
3. Each NavOperation of NavClass becomes a method of the corresponding Ruby class;
4. Each link having NavClass as the source type becomes an attribute of special type “Array”, whose elements are objects of the target type of the link.

The built-in Ruby operators are redefined to handle the expected semantics. For instance, if a new element is inserted in the array that corresponds to a link, this is interpreted as creating a new link instance. Consider for instance the following code

```
schwabe = Professor.find_by_name “Daniel Schwabe”
area = ResearchArea.find_by_name “Hypermidia”
for student in schwabe.advises
  unless student.works_in.include?(area)
    student.works_in << area
  end
end
```

The first statement assigns to variable *schwabe* the object whose *name* attribute is “Daniel Schwabe”, and similarly for *hypermedia*. Next, a loop is performed over each element of the array *schwabe.advises*, which is made out of *student* objects, since the target type for the *Advises* link is *Student*. For each element, it is tested if the *research area hypermedia* belongs to the array *student.works_in*, which contains the *research areas* in which the student works. If this is not the case, the *research area hypermedia* is added to this array, which causes a new link between the *student* and the *research area hypermedia* to be created and inserted in the database.

The use of the DSL is illustrated in the *students_in_area* attribute of class Professor, defined in Figure 10. The intended value for this attribute is defined by “all students advised by professors who work in the same area as the professor in question”. The code in the DSL for this computed attribute is given below.

```
ary = []
for area in self.works_in
  for prof in area.has_professor
    ary << prof.advises
  end
end
ary.flatten!
ary.map { |student| student.name }.uniq.join(",")
```

First an auxiliary variable *ary* is initialized with an empty array. Next, for each area in which the professor (“self”) works, iterate over each professor that works in that area, appending this professor’s advisees to the auxiliary array. At the end of the nested loop, *ary* contains an array of arrays of students, which is then transformed into a flat array through the flatten! method. Finally, a map function over the array of students extracts each student’s name, which is next filtered for duplicates (uniq), and then fed onto the join function to generate a single string with a comma separated list of all the names.

We mentioned earlier that the interface templates is another place where typically code appears, many times interspersed with html, as in the case of JSP or ASP.

Views in HyperDe are defined in a similar fashion, interspersing Ruby expressions with html. These Ruby expressions can include calls for several pre-defined functions, such as “breadcrumbs”, which automatically generates a clickable trace of pages previously navigated by the user. The HTML code below renders the part of the Professor page in Figure 7 where the context information is presented (the gray bar just below the black bar). In this case, the page shows `2/2 Prev |`, which is the relative position of this node in the context (2) and the total number of nodes in the context (2), and a link the previous node.

In the highlighted part, we show the DSL expression that computes this information. This illustrates how SHDM metaclasses are handled by the DSL exactly in the same way as the model classes. The “Context” metaclass has, among others, the “position” attribute, which returns the position of the current node in the context, and the “nodes” attribute, which is an array of the nodes in the context, and “size” returns its size (i.e., the size of the context). Similarly, HyperDe defines the helper functions “context_previous” and “context_next”, which return HTML anchors to the previous and next nodes in the context. Notice that, in this particular case, since the node being exhibited is the last one in the context, the “next” functions returns an empty string.

```
 …
<tr><td width="160"
bgcolor="#99ffcc"> </td><td colspan="2">

<div class="coluna1" style="width:100%;
padding:0.5em;background-color:#eeeeee; margin-
bottom:1em; text-align:right">
<%= @context.position+1 %> / <%=
@context.nodes.size %>

<%= context_previous [ "<a href='%s'>< prev</a>",
:url ] %>
 | 
<%= context_next [ "<a href='%s'>next ></a>", :url
] %>
</div>
</td>
</tr>
```

Consider the interface shown in Figure 8. We can see that it exposes the "Change E-Mail" operation of the node, which is of class "Professor" and inherits it from class "Person" where it is defined (see Figure 10). The code defined for this operation is

```
self.email = controller.params["new_email"]
self.save
```

The first line updates the "email" attribute of the current node, defined by the DSL, to the value returned by the "new_email" form field of the view, as mediated by the controller.

The HTML code that appears in the corresponding view is

```
<input type="text" name="new_email" value="<%=
@node.email %>">
<%= op "change_email", { :label => "Change
Email", :view => "attributes", :update =>
"node_attributes", :label_loading => "wait..." },
[ '<input type=button value="%s" onclick="%s">',
:label, :onclick ] %>
```

This code, which uses the HyperDe pre-defined templates, defines a form with an input field named "new_email", and calls the operation "change_email" when the form is submitted.

Since the metaclasses of SHDM are also part of the DSL, it is possible to define context selection queries in a very general way. For instance, the DSL expression below defines a context that shows all the elements of a subclass of a class name (string) passed a parameter to it:

```
{ |subclass| subclass.constantize.find_all }
```

Specifying a context in this way has the advantage that it is possible to define, for example, an index that will list all subclass names, and each entry would point to the instance of that subclass. Since this definition is independent of the particular subclasses, it remains unaltered each time the application schema is changed by adding new subclasses to the class in question.

For instance, in an online store, we can apply this technique to class "Product", and allow the user to include new subclasses of products without requiring recompilation at all. In practice this achieves a similar effect as having an application framework for online stores, where each specific application has to be redefined each time the classes are changed.

6. DISCUSSION AND CONCLUSIONS

6.1 Evaluation

There are several dimensions in which one can evaluate HyperDe. From the application point of view, two relevant metrics are volume of data, and application complexity. The latter can be measured by the size of the model (i.e., number of classes, contexts, indexes, views, etc…), and the former is related to the number of instances of a given model. Therefore, it is possible to have a very large application that is simple – it has many instances of only few classes, links or contexts, or a small application that is very complex – it has many classes, links or contexts, with few instances of each. Figure 13 shows where HyperDe is situated in this space; it is able to handle complex applications (in the order of several dozens of classes, contexts, links, etc…).

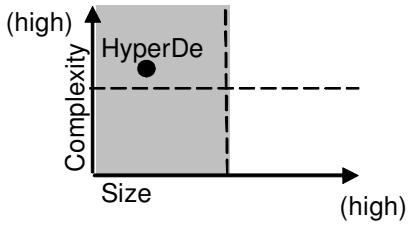

Figure 13. Complexity and Size of Hypermedia Applications

The software infrastructure is based on Ruby, which is an interpreted language, and most likely will be hard pressed to perform for large volume applications, although we have not yet tested its limits.

A second dimension to evaluate HyperDe is development time. The current HyperDe architecture allows very rapid design and prototyping of web applications using SHDM. We have not yet done a formal measurement of development time with and without HyperDe. Nevertheless, we have used HyperDe for several projects, mainly by students, and they were able to design moderately complex applications in the span of one to two days. By comparison, equivalent projects took at least a couple of weeks to be developed without the use of such an environment.

A third dimension under which we have examined HyperDe is flexibility and evolvability. Due to the fact that it is entirely model-based, HyperDe is extremely flexible in allowing and immediately reflecting changes made in the design. In addition, by using the techniques discussed in the previous sections, such as meta-programming, it is possible to design applications that can support schema evolution without change to the code to a very large extent, much greater than in the case of compiled languages.

Due to the speed of development, and the flexibility and evolvability supported by HyperDe, we feel it is well suited for rapid prototyping of web applications.

Finally, a fourth dimension to consider is expressivity and conciseness. We believe that the examples shown in this paper support our claim that the use of models and DSL's allows more concise code, at the proper level of abstraction, when compared to translation-based approaches.

6.2 Related Work

As mentioned earlier, HyperDe is a model-based development environment similar to WebRatio, which is based on WebML; OOWS Tools, which is based on OOWS; ArgoUML, which is based on UWE, and Hera Tool Suite, which his based on Hera. The distinguishing features of HyperDe with respect to all of them are

- The particular model supported;
- The use of meta-models, and making them accessible as part of the design;
- The use of DSL's.

HyperDe is the only environment combining these three aspects.

6.3 Future Work and Conclusions

We have presented HyperDe, an environment that supports the design and implementation of web-based applications, which combines model-based development with domain specific languages. We argued that this combination allows for flexible

and rapid prototyping of applications, as evidenced by several projects conducted in our group and with students.

The development of HyperDe continues, and among the ongoing directions we can cite

- Support for faced navigation (see [8][12]);
- Applying a similar approach to derive DSL's based on RDF schemas, as opposed to SHDM schemas;
- Support for the SHDM abstract interface model as defined in [17];
- Support for adaptation of the application based on several contextual information;
- Experimentation with alternative persistence mechanisms and RDF stores;
- Extension of the SHDM model to allow direct use of ontologies specified with RDF Schema [2] , enriched with navigational information.

Acknowledgement. Daniel Schwabe was partially supported by a grant from CNPq, Brazil.

7. REFERENCES

[1] Beckett, D. RDF/XML Syntax Specification (Revised), W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/

[2] Brickley, D.; Guha, R. V. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[3] BROEKSTRA, J.; KAMPMAN, A. "Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema", Deliverable 10, On-To-Knowledge project, October 2001, http://www.openrdf.org/.

[4] Ceri, S. et al.: Designing Data-Intensive Web Applications. Morgan Kaufmann, 2003

[5] Frankel, D.S.; *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003

[6] Goldman, N. M. Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer, Lecture Notes on Computer Science - The Semantic Web - ISWC 2003, Springer-Verlag Heidelberg, Volume 2870 / 2003 - Outubro, 2003, pp. 850-865

[7] Greenfield, J.; Short, K., *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J.Wiley and Sons Ltd., 2004.

[8] Hearst, M. et al.: Finding the Flow in Web Site Search. Communications of the ACM, 45 (9), September 2002, pp.42-49

[9] Jacyntho, M. D., Schwabe, D. , Rossi, G. A software architecture for structuring complex web applications. Journal of Web Engineering, Vol 1, No 1, (2002).

[10] Koch, N.; Kraus, A.: The Expressive Power of UML-based Web Engineering, 2nd Int. Workshop on Web-Oriented Software Technology (IWWOST´02). CYTED, 105-119, Málaga, Spain. 2002.

[11] Lima, F.; Schwabe, D.: Modeling Applications for the Semantic Web. Proceedings. of the 3rd Int. Conference on Web Engineering (ICWE 2003), Oviedo, Spain, July 2003. Lecture Notes in Computer Science 2722, Springer Verlag, Heidelberg, 2003. pp 417-426. ISBN 3-540-40522-4.

[12] Lima, F.; Schwabe, D.: Application Modeling for the Semantic Web. Proceedings of LA-Web 2003, Santiago, Chile, Nov. 2003. IEEE Press, pp. 93-102, ISBN (available at http://www.la-web.org).

[13] Pastor, O. et al.: "Conceptual Modelling versus Semantic Web: the two sides of the same coin?". Proceedings of WWW2004 Workshop, Application Design, Development, and Implementation Issues in the Semantic Web, New York, 2004.

[14] Rossi, G., Schwabe, D. and Lyardet, F.: Web Application Models Are More than Conceptual Models. Proceedings. of the ER'99, Paris, France, November 1999, Springer, 239-252.

[15] Schwabe, D.; Rossi, G.: An object-oriented approach to Web-based application design. Theory and Practice of Object Systems (TAPOS), October 1998, 207-225.

[16] Schwabe, D. et al.: Design and Implementation of Semantic Web Applications. Proceedings of WWW2004 Workshop, Application Design, Development, and Implementation Issues in the Semantic Web, New York, 2004.

[17] Silva de Moura, S.; Schwabe, D.: Interface development for hypermedia applications in the semantic web. Proceedings. of WebMedia and LA-Web, 2004, Ribeirão Preto, Brazil, October 2004. IEEE Press, pp 106-113.

[18] Thomas, D., Barry, B.M.; "Model Driven Development: The Case for Domain Oriented Programming", Companion of the 18th OOPSLA, ACM Press, 2003, pp. 2-7.

[19] Van Deursen, A.; Klint, P.; Visser, J.; "Domain Specific Languages: An Annotated Bibliography", http://homepages.cwi.nl/~arie/papers/dslbib/

[20] Vdovjak, R., Frasincar, F., Houben, G.J. and Barna, P. "Engineering Semantic Web Information Systems in Hera". In: Journal of Web Engineering, Vol. 2, No. 1&2, p. 3-26, Rinton Press, 2003.