Web Engineering does profit from a Functional Approach

Torsten Gipp and Jürgen Ebert
University of Koblenz-Landau
e-mail: (tgi|ebert)@uni-koblenz.de

Abstract

Founding web site development on models *is the state of the art. This paper aims at showing that* functional specifications *are a powerful means of modelling specific aspects of a web application and that it may be employed to gain overall coherency and an* integrated *set of models. The composition of pages from fragments, the incorporation of content based on queries, the definition of dynamics of the web application, as well as the transformation of pages into appropriate presentation level languages: all these aspects are different models, and they are specified and integrated using a functional language. All models constitute a coherent view on the web site as a whole.*

At the same time, using this functional approach proffers openness and extensibility to incorporate well-established tools and technologies.

1 Introduction

Initial creation and maintenance of websites are two sides of the same coin. A common approach for generating websites and for keeping dynamic websites up-to-date involves separating the different concerns of web site description into different yet integrated documents.

We assume a *model-based view* on web sites: a web site is described by a set of models, each emphasising a particular point of view and at the same time being part of one coherent and consistent 'big picture' of the web site as a whole. In other words, it does not suffice to regard the different models in isolation. Instead, it must be guaranteed that the models are *integrated*. It is important to have a coherent and precise description of all relevant web engineering aspects and of the artefacts (models) that are relied upon. Thanks to the clearly defined integration of the models, which also clarifies that they are *separate concerns*, each model can be tackled individually.

In addition to the models, the *process* of the creation and evolution of a web site has to be considered as well. This is an orthogonal aspect that applies to, and should reflect in, every model.

A prominent focus in our work is *maintenance*. One inherent characteristic of a web site is that it changes and evolves over time, and indeed rapidly so. Therefore, every model created during the development of a web site is potentially changed or even rewritten at any point of time.

One of the core contributions of this paper is to use a *functional programming* approach and a functional language to describe and specify the web site's single pages. We employ the functional programming language Haskell [1] as an example language. The functional specifications are executable and they are used by the run-time system to actually drive the web site. The approach is general enough to provide that the realisation is not constrained to a particular functional language or a particular run-time system. The focus clearly lies on integration, be it on the tools level or on the modelling level.

Relying on functional specifications for the definition of web sites is quite natural insofar as retrieving a web page via HTTP actually *is* a function call. Parameters may be passed in, and the final page is delivered as the result. This result is declaratively defined by a function, which in turn can include calls to other functions so to compose a page from smaller parts. One gains a true amount of coherence since everything in this specification is a function.

At the same time, this approach fosters modularity. The granularity of the fragmentation can be chosen with all flexibility of the functional language, and it can also differ on a page-to-page basis. This allows for doing template-based page generation, which we exploit by proposing an example template mechanism to build pages with a consistent layout. This mechanism can easily be substituted or extended if more complexity is needed.

A functional specification is a formal description. At the same time, it is an implementation that can be executed. This advantage can be exploited for prototyping and even for simulation of web sites. The specification represents a high-level view on the page definitions, but also lends itself to 'drilling down' due to its inherent rigourousity.

Common maintenance activities, e.g., established configuration management disciplines, can be applied almost naturally to the functional specifications.

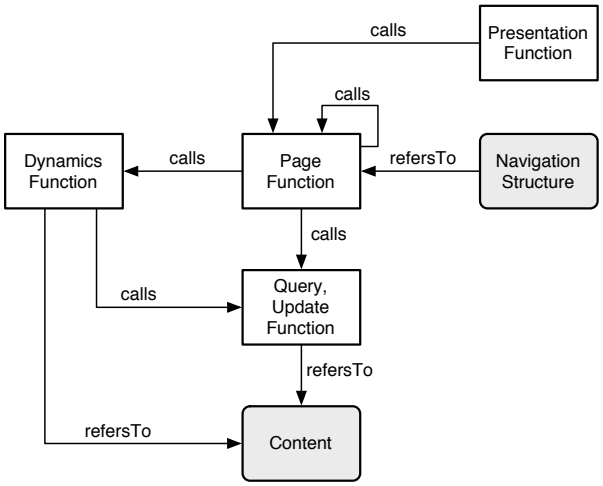

Figure 1. Artefact integration

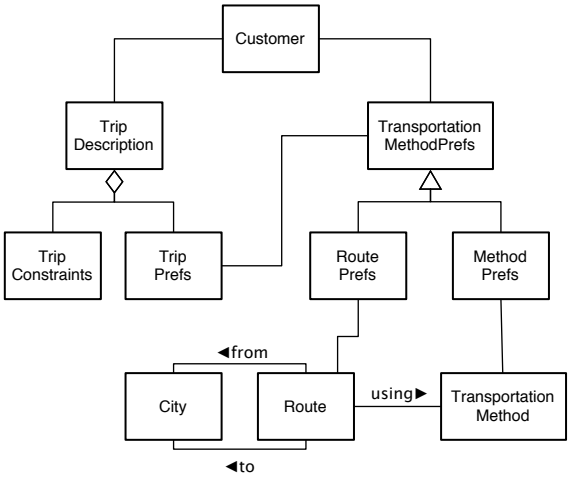

Figure 2. Content model for the TAS example

As suggested in the call for papers, the approach will be presented by using the travel agency system (TAS) as an example. Cf. [2] for a description of this system.

The remainder of the paper is structured as follows. The following section 2, backed up by the TAS example, introduces the models that yield the 'big picture,' giving a detailed explanation of the application of the functional specifications. Section 3 emphasises the model integration, section 4 gives an overview of related work, and section 5 concludes by summarising the core items of this text.

2 Models for the TAS example

According to the separation of concerns mentioned in the introduction, we provide a set of core models that, taken together, capture the application domain and its projection into a set of web pages. We consider: the content, the navigation structure (site map), the pages (navigation objects), queries and updates, the presentation, and the dynamics. The models are *integrated*, and sections 2.1 to 2.6 each describe one model and its integration with the others. Figure 1 visualises the interdependency of the models. After introducing the single models, section 3 will re-focus on their integration.

2.1 Content

The content model captures the concepts of the application domain and their relationships. A UML class diagram is used to write down the model (see fig. 2). For demonstration purposes, it is sufficient to consider a subset of the classes only. The diagram in figure 2 only contains classes that deal with the description of a trip and the storage of a customer's preferences in terms of transportation methods and routes.

The UML diagram describes the structure of the repository that keeps all contents. A query facility (see section 2.4) allows the extraction of information for the inclusion in the web pages.

2.2 Navigation Structure

The *navigation structure* determines the relation of the single pages with respect to the hyperlinks between them. It is modelled with a visual language that provides special features, like distinct page types and authorisation-dependent navigation, thus defining the *site map*. This language has been successfully applied in some of our web engineering projects.

Figure 3 visualises the site map for the TAS. The *primary* navigation structure, given by the solid arrows, defines a tree of page nodes. This tree assigns a unique path to every page. Also, this tree structure is easy to communicate to a web site visitor, who can create a mental image of the site map fairly quickly, which in turn is a very important ergonomic feature.

The *secondary* navigation structure is visualised by dashed arrows. They represent arbitrary links between pages, without heeding the tree structure.

There are four different types of pages, visually differentiated by four different page icons (cf. the legend in fig. 3). A lightning bolt marks a page as being *dynamic*, i.e., as a page whose relevant content is calculated (and thus potentially varies) at the time of access. In contrast, the content of *static* pages does not change at run-time. The classification of a page as being either static or dynamic is not necessarily unambiguous, because the definition of 'relevant content' is subject to interpretation. The distinction merely serves communicative purposes during modelling. There are no consequences on the implementation level.

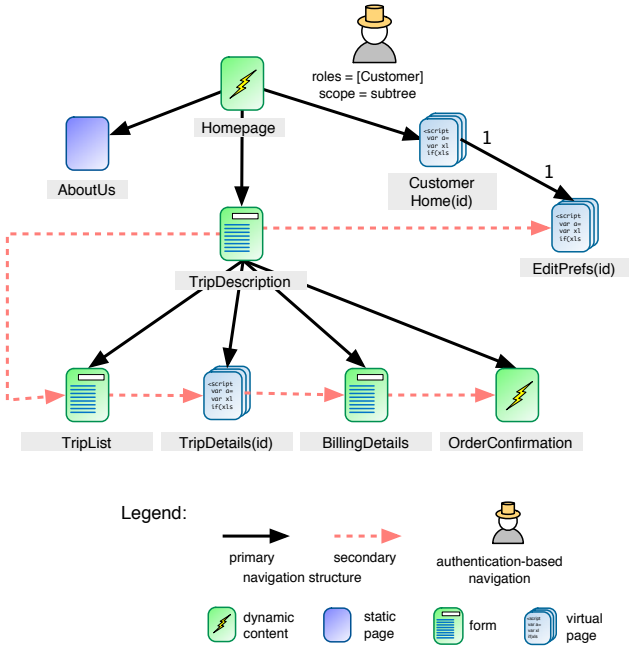

Figure 3. Navigation structure

Pages providing a form to let a web site visitor enter some data can be distinguished by a corresponding *form* icon.

A small piece of script code on a stacked page icon signifies a *virtual* page that is computed by a script. In contrast to the 'lightning bolt' pages with dynamic content, the script-generated pages are *entirely* calculated by a set of parameters, where one (the first) parameter defines the name of the page. The virtual pages do not exist under a pre-defined identifier, like the pages with dynamic content do. They are rather created and evaluated on-the-fly, every time the page is called. We will use the term *instance* to talk about concrete virtual pages. There is one instance for each possible identifier.

These four basic web page flavours can also be mixed on one page. A virtual, a static, or a dynamic page can contain a form (or more than one). Since non-dynamic virtual pages do not make much sense – because this would mean that every instance looked the same and did not make use of the identifying parameter – the lightning bolt adornment will not be applied to virtual page icons, and virtual pages will count as *always* being dynamic.

Technically, pages of all four page types are defined by a page function, and every page function has the *same* signature. Therefore, the page type chosen in the site map diagram is of no relevance implementation-wise (see section 2.3).

The navigation structure diagram may also contain information on *authorisation-dependent navigation*. In the example, the primary link to CustomerHome(id) is annotated with a role-icon. It states that a web site visitor must possess the role Customer in order to access the page. The scope=subtree declaration expands this constraint to the whole subtree rooted at this page. The alternative value thisPage for scope would prohibit this expansion. The actual mechanism for checking the authorisation of a given user, a given action and a given object is intentionally left open in our approach. We can encompass any matrix-based scheme that assigns any number of permissions to perform actions to a list of roles. Thus, the actual mechanism of the implementation platform can be used here.

The diagram can also capture the *multiplicity* of links to or from virtual pages. This is useful because virtual pages are like classes in that they represent a set of instances. Thus, we adopted a subset of the UML's multiplicity symbols to lay down how many instances may be connected. In figure 3, each CustomerHome(id) instance is connected to exactly one instance of EditPref(id).

The actual checking of constraints and of the authorisation is contained in the associated page functions. They also contain the definition of the links for the secondary navigation structure. Thus, we can employ the full power of the underlying functional language to provide conditional links, whose behaviour or mere existence depends on the system state and other context information.

2.3 Pages

The pages that constitute the navigation space, i.e., the set of objects or nodes that can be visited, are called *navigation objects*. Each page is defined by a *page function* that returns an abstract regular structure which can be mapped to a concrete renderable page description using a presentation level language. Every time a page is accessed, the corresponding page function is evaluated, the result is transformed into the appropriate presentation language, and the result is sent to the client (see section 2.5).

We define a data type, the *abstract page description* (APD), that allows for defining a page on an abstract level in terms of nested, labelled, and attributed elements (analogous to XML). Here is the definition of this data type (in Haskell):

```
data APD =
      Text String
    | Element Name Attrs ElementList
    | Link Name Attrs ElementList Identifier Params
    | Form Name Attrs ElementList Identifier Params
    | Field Name Attrs String
    | Empty
```

There are six constructors for the APD type. An APD term can be a simple text node (Text); an element (Element) with a name, a list of attributes, and a list of child terms; a link or a form (Link and Form) with a name, a list of attributes, a list of child terms, the identifier of the destination page, and a list of parameters that should be passed to this page; a

field in a form (Field) with a name, a list of attributes and a default field content; or it can simply be empty (Empty).

This declaration uses some type synonyms (syntactical abbreviations):

```
type Name = String
type Attrs = [(String, String)]
type ElementList = [APD]
type Params = Attrs
type Identifier = String
```

The name of an element is a string. Attributes and parameters are modelled as lists of key/value-pairs. Elements as well as forms can contain child nodes, so they use ElementList as a container for a list of arbitrary APD structures.

The type PageFunc is the function type for pages, mapping parameters to an APD structure. It is used for every page definition.

```
type PageFunc = Params → APD
type Id2PageFunc = Identifier → PageFunc
```

Links and form destinations are defined in terms of function identifiers. In this context, a function of the type Id2PageFunc maps identifiers to real page functions. This implies that links are represented by terms in an APD structure, attached with a reference to the page function they link to. This allows for link consistency checks.

First Example. As an example page function, consider the following (simplistic) definition:

```
greetingPage :: PageFunc
greetingPage [("name", name)] =
    Element "pageheading" []
      [ Text "Hello"
      , Element "paragraph" [] [Text name]
      ]
```

This page is to be called with one parameter (called name) and results in an APD term that consists of a root element pageheading, which comprises a text element and a paragraph element. The latter finally produces another text element which corresponds to the value of the parameter that was passed to the page function. This is a first, albeit dull, demonstration of how to access page parameters and how to incorporate dynamic content.

Example with query. In general, the page functions are the place where we define the incorporation of the actual content into the pages. We suggest using a set of functions that encapsulate *queries* on the content repository. Since this is a separate level of abstraction, we treat these queries as a model of its own. Section 2.4 goes into more detail.

As an example, incorporating a list of a customer's transportation method preferences into the TripDescription page is done with a function

```
getMethodPrefs :: String → [String]
```

Given a customer's id as parameter, it returns a list of transportation method names. The actual query may be arbitrarily complex, but this does not matter at this point.

In the corresponding page fragment that calls this function, the delivered result is transformed into a list structure so it can be presented to the web site visitor:

```
-- generates a list of a customer's preferences
tasPreferencesList :: PageFunc
tasPreferencesList params =
  let
    methodPrefs = getMethodPrefs                                    (1)
      (fromJust (lookup "customerId" params))
  in
    foreach
      methodPrefs
      (Element "list" [] [])
      (λ(methodName) →
        Element "item" [] [ Text methodName ]
      )
```

This listing shows the page function that builds an APD for the presentation of a customer's preferences (only the transportation methods are considered). The page function receives a customer id as its parameter. The main body of the function calls getMethodPrefs in the line marked (1) and then iterates over the resulting list of transportation method names. The iteration is achieved via the foreach construct. This is a simple iterator that applies a given function to each element in a given list and returns an APD with the result. Here, we can see how powerful the approach can get in respect to the specification complexity: iterators (and other control flow constructs, like higher level functions) can be defined and easily applied, thus lifting the specifications to a higher level of abstraction and allowing for more dynamics.

Example with template. For standard applications it appears to be appropriate to divide pages into several areas. Figure 4 visualises such a general page layout by showing a page template with *slots*. The slots are the places where the actual content is to be put in.

The presentation of a page as a function can be based on this template. As such, it serves as an additional abstraction of a page: the overall layout of the page in terms of its presentation can be defined without caring about the concrete content. The presentation of the content is again defined independently.

A page template is a function $f : \mathsf{APD} \rightarrow \mathsf{APD}$. This signature emphasises that it is a *filter* that can be interposed in the transformation process (in the sense of a pipe/filter architecture). The functional approach allows for employing any number of these filters, resulting in further abstraction levels

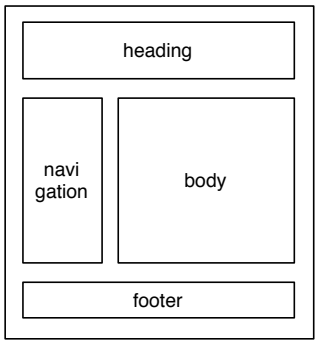

Figure 4. Page template with named *slots*

for pages. The levels can be established in correspondence to the complexity of the particular application at the time of modelling. There is no need to change the definition for the APD. The actual levels employed can even vary from page to page.

In order for a template to work as a filter, the *input* has to be an APD as well. Therefore, each filter traverses the given APD and searches it for content fragments that are marked with slot identifiers, thus signalling where to put them in the resulting page. That is, one can regard pages that make use of page templates as containing a *mapping* from slot identifiers to slot content. Calling a page template then involves creating the output APD and inserting the slot contents at the defined places by using this mapping.

We gain a further degree of coherence by defining the mapping as an APD as well. A page using a particular page template contains elements that have the same name as the slots defined in the page template. The children of these elements are then put at the appropriate place in the resulting APD.

The following listings show the definition of the TripDescription page. This page uses a page template (tasMainTemplate) whose Haskell code is given further below. The page template tasMainTemplate is called with an element named slots as its argument. This element serves as a container for the slot content mapping. The adherence to this convention is expected by the page template, which searches the given APD for the slots it is responsible for.

```
tasTripDescription :: PageFunc
tasTripDescription params = tasMainTemplate
  (Element "slots" []
    [ Element "heading" [] [Text "Trip␣Description"]
    , Element "navigation" [] [ tasNavigation params ]
    , Element "body" []
      [ (tasTripDescriptionForm [])
      , (tasPreferencesList params)
      ]
    , Element "footer" [] [ Text "Footer" ]
    ])
```

Here is the main template:

```
-- Main template. Returns a complete HTML page.
tasMainTemplate :: APD → APD
tasMainTemplate (Element "slots" a es) =
  Element "html" a
    [ Element "head" [] -- HTML head
      [ Text "TAS" ]
    , Element "body" [] -- HTML body
      [ Element "div" [("class", "heading")]
          ( -- pageheading slot
            if (length headingContent) ≠ 0 then
              headingContent
            else
              [ ]
          )
      , Element "div" [("class", "body")]
          ( -- body slot
            if (length bodyContent) ≠ 0 then
              bodyContent
            else
              [ ]
          )
      -- ( navigation slot, footer slot omitted )
      ]
    ]
  where
    headingContent =
      (filter (isElementWithName "heading") es)
    bodyContent =
      (filter (isElementWithName "body") es)
```

The function isElementWithName used in the **where** clause is a boolean function that returns true if, and only if, a given element has the name given.

The other functions necessary to constitute the complete TripDescription page are given below, to present a rather complete example. Note that they only build a very 'naked' version of that page. The page fragment building the navigation tree is omitted for the sake of brevity. The tree is defined by the primary navigation which is laid down in the navigation structure diagram.

```
tasNavigation :: PageFunc
tasNavigation _ = Empty

tasTripDescriptionForm :: PageFunc
tasTripDescriptionForm [] =
  let
    originCities = foldr
        (λa b → a ++ "\n" ++ b) "" getOriginCities
    destCities = foldr
        (λa b → a ++ "\n" ++ b) "" getDestCities
  in
    Form "TripDescriptionForm" []
      [ Field "originCity"
          [("type", "optionlist"), ("values", originCities)] ""
      , Field "destCity"
```

```
        [(”type”, ”optionlist”), (”values”, destCities)] ””
    , Text ”Date␣of␣departure:”
    , Field ”dateOfDeparture” [] ””
    , Text ”Date␣of␣return:”
    , Field ”dateOfReturn” [] ””
    ]
    ”tasTripDetails” []
```

The get...-functions used here encapsulate queries to the content repository. For the example assume that getMethodPrefs returns [”Train”, ”Car”], getOriginCities returns [”Paris”], and getDestCities returns [”New␣York”, ”Rio”, ”Tokyo”] (section 2.4 will present an example query).

Besides the isElementWithName function, a whole set of auxiliary functions has been defined in order to make the definitions shorter and easier to maintain. Examples for tasks they perform is working with parameters lists, traversing APDs to find specific elements (especially links and forms), and displaying debug information.

One of the debugging functions prints a textual representation of an APD. Calling it with the results of the TripDescription page yields:

```
Element ”html” []
 [Element ”head” [] [”TAS”]
 ,Element ”body” []
  [Element ”div” [(”class”,”heading”)]
   [Element ”heading” [] [”Trip␣Description”]]
   ,Element ”div” [(”class”,”body”)]
    [Element ”body” []
     [Form ”TripDescriptionForm” []
      [Field ”originCity” [(”type”,”optionlist”),
            (”values”,”Paris\n”)] ””
      ,Field ”destCity” [(”type”,”optionlist”),
            (”values”,”New␣York\nRio\nTokyo\n”)] ””
      ,”Date␣of␣departure:”
      ,Field ”dateOfDeparture” [] ””
      ,”Date␣of␣return:”
      ,Field ”dateOfReturn” [] ””
      ]
      <page: ”tasTripDetails” []>
     ,Element ”list” []
      [Element ”item” [] [”Train”]
      ,Element ”item” [] [”Car”]
      ]
 ]] ]]
```

Figure 5 visualises this result, hinting at the final presentation of the page. The abstract language used is quite close to XHTML already, but this is not a necessity. The final transformation into a concrete presentation level language is performed in a separate step, described in section 2.5.

2.4 Queries and Updates

Since the content model is given in terms of classes and relationships, it is possible to use almost any kind of representation for the underlying content repository. In our implementation we rely on a graph repository which keeps the data as *TGraphs* [3], i.e. typed, attributed and ordered directed graphs. This repository supplies a functional querying language that allows for accessing the graph at run-time, including updates.

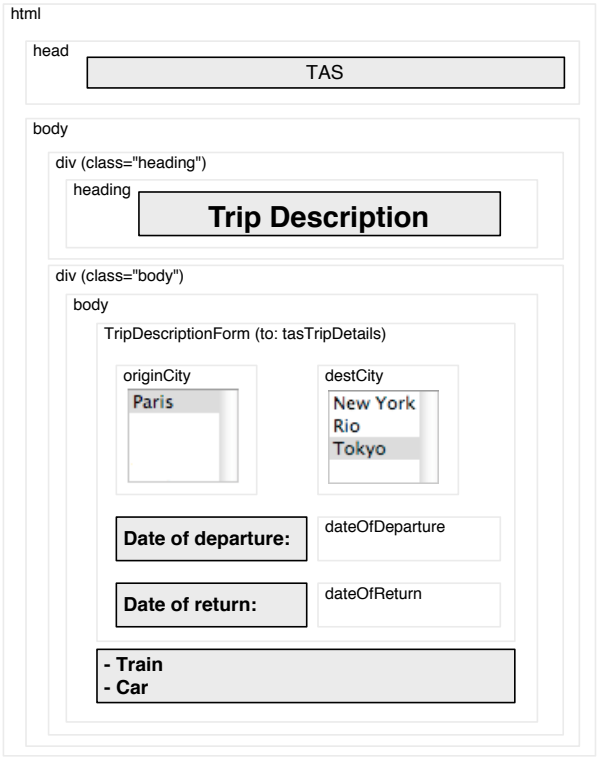

Figure 5. Visualisation of the abstract page structure for the TripDescription page.

The queries used for content integration can be of arbitrary complexity. While the query API may well be very simple, the full strength of the functional language can be used to work on the query results before incorporating them into the pages. Thus, each query is a proper function in its own right. This allows for defining arbitrarily complex *views* on the content. We prefer this amount of flexibility in favour of only allowing simple one-to-one mappings between the content model and the hypertext model. The goal is to keep the page functions as simple as possible and to avoid a too strong intermixing of content accessing functionality into the page definitions. This is perfectly achieved by using query functions that return simple objects like single ‘records’ or lists of records. The page functions then simply access the records or iterate over a list. Any necessary computation is encapsulated inside the query functions.

Consider, as an example, the following query that retrieves the list of all available cities:

```
queryAllCities :: AttributedGraph → [String]
queryAllCities g =
  nodesToValues
    g
    (λ lbl → getValue lbl "id")
    (query g (nodes g) [ constrainByType "City" ])
```

Without diving into the implementation details, we can note that this function returns a list of strings, given a concrete Graph g, by first selecting all nodes that are of type City, and then mapping a function that extracts the value of the id attribute over this list of nodes, resulting in the desired list of city names.

2.5 Presentation

The presentation model is given by defining one or more mappings (presentation functions) from an APD to the corresponding presentation level language. In the case of a web application that is to be delivered via HTTP and is destined to be rendered by a user agent that understands XHTML, a simple transformation of the regular APD into XHTML can be implemented as a Haskell function. Alternatively, the APD could be converted to any other XML dialect first, and subsequent transformations may be done with technologies like XSLT. All conceivable possibilities are open at this point, and the approach can be easily adapted to a great number of run-time systems. Note that the actual transformations can be selected at run-time, even on a page-to-page basis, or according to *context* information. This opens the path to customisation, personalisation, and multi-mediality.

For our current implementation, we use the *Zope* [4] web application server and we integrated a Haskell interpreter (Hugs, [5]) that evaluates page functions on-the-fly. The page functions (via a presentation function) return complete XHTML documents which can be delivered without further transformation.

2.6 Dynamics

The behavioural aspect of the TAS can be modelled elegantly by looking at the different functions that have to be carried out by the actors. Thus, the 'business logic' is broken down into well-specified functions that can be glued together in the page function definitions. Table 1 lists three example functions. It states the function provider and the name of the function in the first line, followed by a short description of its semantics. The rows marked 'In' and 'Out' describe the input and output parameters, respectively.

Since the page functions are executable formal specifications, the dynamics of the system can be subjected to simulation and testing. One can devise test cases by anticipating the results of relevant functions and test them by calling them with appropriate arguments.

Table 1. Function list

PTA.checkWellformedness	
	Checks a trip description for plausibility (e.g., return date lies before departure date)
In	trip description (TripDescription)
Out	true, if trip description is well-formed, i.e., plausible; false, otherwise
PTA.prepareTripList	
	Processes a well-formed trip description and presents a list of matching trips
In	trip description (TripDescription)
Out	ordered list of trips that match the trip description
TC.bookTemporarily	
	Temporarily books a transportation service, if possible. Reports any failure.
In	service request (Route with associated origin, destination, and transportation method; date of departure)
Out	confirmation of success (true or false)

PTA: personal travel agency; TC: travel company

3 Integration

The integration of the different models just described in section 2 is achieved by making functions call one another and by referring to the concepts from the content model in the functional specifications. Figure 1 depicts these relationships. A page function declaratively specifies what one navigation object consists of, thus being a model for it. At the same time, it is a callable function that returns the actual page upon evaluation. In other words, the *functions are executable models*. The page functions call query and update functions as well as dynamics functions, which makes them the integral and pivotal glue that holds the system together. They also implement the secondary navigation structure by including corresponding links.

By accessing instance data that complies to the content model, the functions refer to concepts from the application domain. Functional requirements are captured by specific dynamics functions that also contribute to building higher-level, integrating and integrated functions that can be used in the other specifications.

We showed how a complete website can be described by six different kinds of documents. Content is modeled by a conventional UML class diagram and the site's structure is modeled by a site graph. The site graph is written in a visual language that differentiates between four different page types (static, dynamic, form-based, and virtual pages) and allows to include authorisation-dependent navigation constraints.

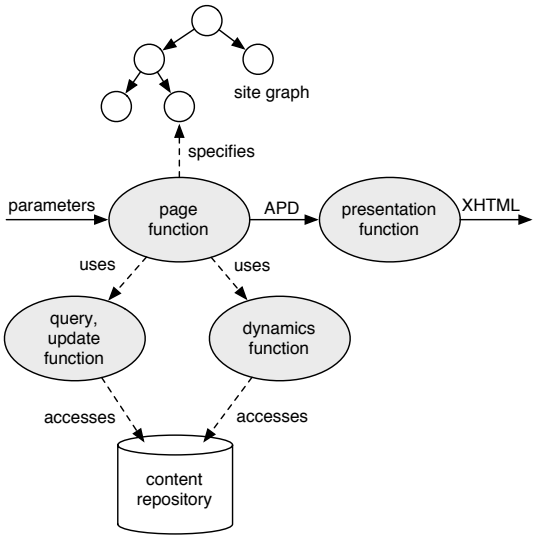

Figure 6. Artefact integration: data flow

All other information necessary to fully define the web site, i.e., the pages themselves, including all page fragments that are part of them, the queries supplying content information to the pages, the rendering of the pages for presentation, and the dynamics of the pages, are described as functions using a functional language.

All this effectively guarantees a high level of coherency. Figure 6 visualises the integration of the different artefacts, focussing on the flow of data, from the receipt of parameters to the production of a document in the desired output language. The page functions are the integrating unit, using queries, updates, and dynamics functions to define the incorporation of data into the pages and the execution of 'business logic'. The page functions also specify the site graph by creating APD terms that represent links.

4 Related Work

Relying on models for describing and specifying web sites has quite a long tradition. Overviews and comparisons of the most prominent approaches are given e.g. in [6], [7], and [8]. The approaches can be very coarsely classified by their 'foundations': some focus on object-oriented models, others rely on entity-relationship models, and again others put documents into the center of interest. The most influential 'schools' are the graph-based Strudel approach [9], the TSIMMIS project [10], the ER-based RMM [11], Araneus [12], HDM [13] and OOHDM [14], WebML [15], and UWE [16].

Significant effort has been put in developing and describing diverse methodologies for web site generation, of which none, to our knowledge, relies as much on functional specifications as we do. We envision a synergetic potential for the integration of our findings into existing approaches, or, vice versa, the integration of selected parts of the aforementioned approaches into ours. This vision was the reason for our approach being as abstract and as extensible as possible. The idea of integrating the models by making them functions, which is unique to our approach, clearly works best when *all* models are specified as functions.

It is interesting to compare the various notations used in the respective approaches. Some approaches rely on proprietary notations for some of the diagram types, especially for the hypertext models. A majority of the current approaches employs the UML (and its extension mechanisms) for the notation of diagrams. The main reasons stated for using UML are the availability of tools ([17, p. 2]), the fact that the UML is well-documented ([18, p. 2]), and the coherence gained by using UML for a web application that is connected to other systems that are already modelled using UML ([19, p. 64]). As of today, one can state that using UML class diagrams for the notation of entity-relationship views simply is standard practice.

It is not just recently that the community realises that *aspect-orientation* can very well be applied to web engineering problems. Many cross-cutting aspects have been identified (principally: authorisation, contextuality), and the models are defined in a way to allow for the inclusion of aspects (e.g., [20]). Our approach nicely fits into this line, as the functions offer well-defined cut points.

Our approach is based on functional specifications. We aim at integrating the advantages of this 'way of thinking' into existing web engineering practice. To the best of our knowledge, only very little effort has been put into this direction. Producing HTML and XML with a functional language in a type-safe way is, e.g., investigated in [21], [22]. A way of representing graphs in Haskell is proposed in [23], accompanied by a working implementation.

5 Summary and Conclusion

This text presented a coherent approach to web engineering by relying on functional specifications. Page functions are the formal foundation upon which whole web sites can be built.

At the same time, the approach is general and open enough to incorporate existing tools and technologies, like graph querying for the inclusion of content, web application servers or web content management software that provide session management and access control, or configuration management software to take care of the artefacts produced during development and evolution. Testing and simulation can be done by executing the specifications.

Note that abstracting pages, the dynamics, queries, and updates as functions effectively paves the way to the encapsulation of selected functions as *web services*.

Exploiting the ease of use and extensibility of the functional approach, we introduced a simple template mechanism to describe pages on a higher level of abstraction.

Once a consolidated library of functions is available, creating the functional specifications is not very hard work. However, there is also room for further improvement, especially when thinking of software tools that let the web engineer work in a more visual environment. These tools then deliver the functional specification as their *output*. It might make sense to constrain the expressive power of these tools, so as to be able to maintain a two-way consistency between the generated functional specifications and the visual documents shown to the user.

We regard our approach as a contribution that shall serve as a foundation, paving the ground for building upon and exploiting its possibilities.

References

[1] J. Peterson and O. Chitil, "The Haskell Home Page." http://www.haskell.org/, Dec 2004.

[2] "A travel agency system." http://www.lcc.uma.es/~av/mdwe2005/TheTASexample/, May 2005.

[3] J. Ebert and A. Franzke, "A Declarative Approach to Graph Based Modeling," in *Graphtheoretic Concepts in Computer Science* (E. Mayr, G. Schmidt, and G. Tinhofer, eds.), no. 903 in LNCS, (Berlin), pp. 38–50, Springer, 1995.

[4] "Zope.org." http://www.zope.org, May 2005.

[5] "Hugs 98 Web Site." http://www.haskell.org/hugs/, August 2004.

[6] N. Koch, "A comparative study of methods for hypermedia development," Technical Report 9905, Ludwig Maximilians-Universität München, November 1999.

[7] P. Fraternali, "Tools and approaches for developing data-intensive Web applications: a survey," *ACM Computing Surveys*, vol. 31, no. 3, pp. 227–263, 1999.

[8] G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger, eds., *Web Engineering: Systematische Entwicklung von Web-Anwendungen*. Heidelberg: dpunkt.verlag, 2004.

[9] M. Fernández, D. Florescu, A. Y. Levy, and D. Suciu, "Declarative specification of Web sites with Strudel," *VLDB Journal*, vol. 9, no. 1, pp. 38–55, 2000.

[10] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom, "The TSIMMIS project: Integration of heterogeneous information sources," in *16th Meeting of the Information Processing Society of Japan*, (Tokyo, Japan), pp. 7–18, 1994.

[11] T. Isakowitz, E. A. Stohr, and P. Balasubramanian, "RMM: A methodology for structured hypermedia design," *Communications of the ACM*, vol. 38, no. 8, pp. 34–44, 1995.

[12] G. Mecca, P. Merialdo, and P. Atzeni, "Araneus in the era of XML," *IEEE Data Engineering Bulletin*, vol. 22, pp. 19–26, September 1999.

[13] F. Garzotto, P. Paolini, and D. Schwabe, "HDM – a model-based approach to hypertext application design," *ACM Transactions on Information Systems*, vol. 11, no. 1, pp. 1–26, 1993.

[14] D. Schwabe and G. Rossi, "The object-oriented hypermedia design model," *Communications of the ACM*, vol. 38, no. 8, pp. 45–46, 1995.

[15] S. Ceri, "Web Modeling Language (WebML): a modeling language for designing Web sites," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 33, no. 1–6, pp. 137–157, 2000.

[16] A. Knapp, N. Koch, G. Zhang, and H.-M. Hassler, "Modeling business processes in web applications with ArgoUWE," in *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings* (T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, eds.), vol. 3273 of *LNCS*, pp. 69–83, Springer, 2004.

[17] E. Gorshkova and B. Novikov, "Exploiting UML extensibility in the design of web information systems," in *Proc. Fifth International Baltic Conference on Databases and Information Systems*, (Tallinn, Estonia), pp. 49–64, June 2002.

[18] N. Koch, A. Kraus, and R. Hennicker, "The authoring process of the UML-based Web engineering approach." http://www.dsic.upv.es/~west/iwwost01/files/contributions/NoraKoch/Uwe.pdf, June 2001. (on-line).

[19] J. Conallen, "Modeling Web application architectures with UML," *Communications of the ACM*, vol. 42, no. 10, pp. 63–70, 1999.

[20] G. Zhang, H. Baumeister, N. Koch, and A. Knapp, "Aspect-oriented modeling of access control in Web applications," in *Proc. 6th Int. Wsh. Aspect Oriented Modeling (AOM)*, (Chicago, Illinois, USA), March 2005.

[21] P. Thiemann, "Modeling HTML in Haskell," in *Practical Aspects of Declarative Languages* (E. Pontelli

and V. S. Costa, eds.), vol. 1753 / 2000, (Second International Workshop, PADL 2000, Boston, MA, USA), p. 263, Jan 2000.

[22] P. Thiemann, "A typed representation for HTML and XML documents in Haskell," *Journal of Functional Programming*, vol. 12, pp. 435–468, July 2002.

[23] M. Erwig, "Inductive graphs and functional graph algorithms," *Journal of Functional Programming*, vol. 11, no. 5, pp. 467–492, 2001.