Chapter 3

The Presentation Generation Phase of Hera

Hera is a model-driven methodology for designing Semantic Web Information Systems (SWIS). The presentation generation phase of the Hera methodology builds a Web presentation for some given input data. Based on the principle of separation of concerns, Hera defines models to describe the different aspects of a SWIS. These models drive the specification of the data transformations used in the implementation of the Hera presentation generation phase. The Hera presentation generation phase has two variants: a static one that computes at once a full Web presentation, and a dynamic one that computes one-page-at-a-time by letting the user influence the next Web page to be presented. The dynamic variant proposes, in addition to the models from the static variant, new models to capture the data resulted from the user's interaction with the system. The implementation of the static variant is based on XSLT data transformations and the implementation of the dynamic variant is based on Java data transformations.

3.1 Introduction

Hera is a SWIS design methodology. It proposes design steps that, based on the separation of concern principle, specify different aspects of a SWIS. These specification aspects are given by models that have graphical representations. The implementation of a SWIS using the Hera methodology is based on data transformations driven by Hera models. Hera has its origins in the RMM design methodology [Diaz et al., 1997]. Differently than RMM, Hera specifies also other features of a SWIS like the look-and-feel aspects, the user interaction with the system, or the presentation adaptation.

Figure 3.1 shows the main phases in Hera: *data collection* and *presentation generation*. The Hera methodology comes also with a straightforward implementation in which the Hera's main phases and the design steps corresponding to these phases are naturally

mapped to components in a pipeline software architecture. We point out that the software based on this architecture is just one of the possible implementations of SWIS given the specifications required by the Hera methodology.

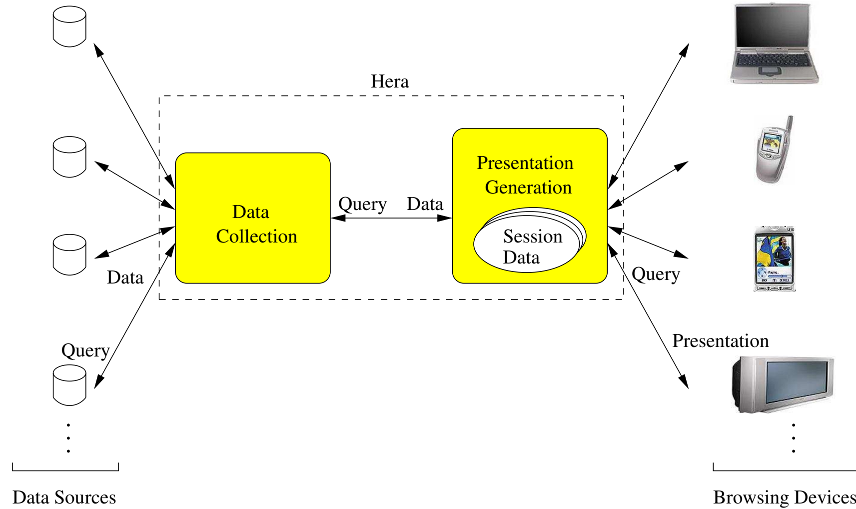

Figure 3.1: Hera's main phases.

The data collection phase helps to make the data available from different sources, such that in response to a user query a data result set is obtained. In this phase of the process the integration model is defined that maps data from the different sources to a common data representation. This mapping is needed whenever for a given query the instances that compose the query result need to be retrieved. The data collection phase is outside the scope of this thesis. More information on this phase can be found in [Vdovjak et al., 2003].

The presentation generation phase builds a hypermedia presentation for the retrieved data. It is based on a sequence of data transformations driven by several models. These models depict different application aspects that are relevant in this process: what is the domain of the application, what is the navigation structure for data from this domain, how to arrange and style the data on the user's display, and how can we tailor the generated presentation based on user preferences and user browsing platform. As can be seen from Figure 3.1 the generated hypermedia presentations can target different platforms like PC, WAP phone, PDA, etc.

The presentation generation phase has two variants: a static one in which the user is unable to change the content of the generated hypermedia presentation and a dynamic one which considers the user interaction with the system in the process of building the next hypermedia page. In the static variant all pages are generated before the user browses the presentation and in the dynamic variant one page is generated-at-a-time during the browsing.

In order to better support the description of Hera's presentation generation phase we use a running example based on real data coming from the painting collection in a museum, the Rijksmuseum in Amsterdam.

The remainder of this chapter is structured as follows. Section 3.2 explains why we chose RDF as a model representation language. Section 3.3 presents the static presentation generation phase of Hera. Section 3.4 presents the dynamic presentation generation phase of Hera. Section 3.5 concludes the chapter and presents future work.

3.2 RDF(S)

For the Hera specifications RDF(S) [Lassila and Swick, 1999; Brickley and Guha, 2004] is used. RDF(S) is the foundation language of the Semantic Web. There are several reasons for choosing RDF(S): it is flexible (it supports schema refinement and description enrichment), it is extensible (it allows the definition of new resources/properties), and it fosters Web application interoperability (it provides a framework to describe in a uniform way the data semantics). As RDF(S) doesn't impose a strict data typing mechanism it proved to be very useful in dealing with semi-structured (Web) data. On top of RDF(S) high-level ontology languages (e.g., DAML+OIL [Connolly et al., 2001], OWL [Bechhofer et al., 2004]) are defined, which allows for expressing axioms and rules about the described classes giving the designer a tool with larger expressive power. We believe that choosing RDF(S) as the foundation for describing models enables a smooth transition in this direction.

Hera models are described in RDFS. An RDFS vocabulary is developed for each model in order to define the model's concepts (which are the classes and properties to be used in a model). Model instances have an RDF representation which are validated against their corresponding schema (model). Having such standards to express models enables the model reuse between different applications. The use of RDFS allows us also to reuse existing RDFS vocabularies for expressing for example domain models or user profiles.

In some applications built with Hera we successfully reused the domain model developed for museum descriptions in the TOPIA (Topic-based Interaction with Archives) project [Rutledge et al., 2003] and the User Agent Profile (UAProf) [Wireless Application Protocol Forum, Ltd., 2001], a Composite Capability/Preference Profiles (CC/PP) [Klyne et al., 2004] vocabulary for modeling device capabilities and user preferences.

3.3 Presentation Generation (Static)

The typical structure of the static variant of the presentation generation phase is given in Figure 3.2 in terms of three layers: the conceptual layer defines the content that is managed in the SWIS, the application layer provides the navigation structure on the data, and the presentation layer gives the presentation details that are needed for the generation of the hypermedia presentations on a concrete platform. As can be noted from Figure 3.2, in the static variant for the presentation generation phase the whole Web presentation is produced at once in response to a user query.

The presentation generation phase distinguishes the following steps: the conceptual design, the application design, the presentation design, and the implementation. Each

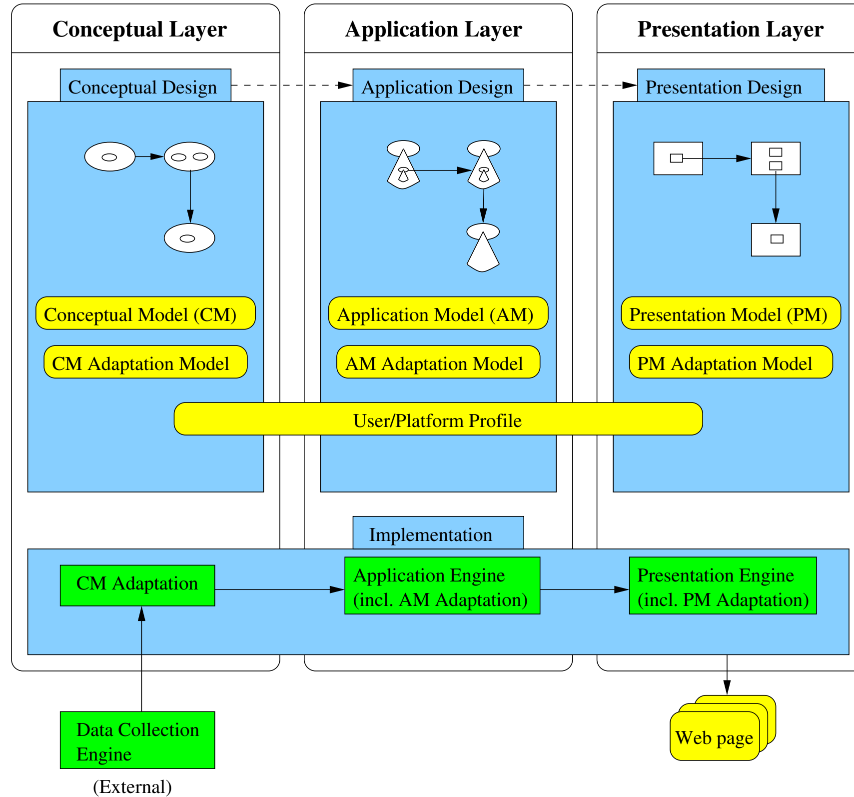

Figure 3.2: Presentation generation phase (static).

design step produces appropriate models that capture the design aspects specific to this step. A model uses concepts from a model-specific vocabulary. In order to ease the specification of the models the model concepts have associated graphical representations. In this way a model can be showed as a diagram to facilitate the designer development and understanding of a certain model.

Adaptation [Frasincar et al., 2004] is not seen as a separate design phase because this process is distributed through all the previously identified design steps. In the adaptation design the user/platform profile (UP) is defined, i.e., it is determined which are the user preferences and platform characteristics that can influence the Web presentation before the user starts the browsing session. The adaptation model specifies adaptation conditions (Boolean expressions) used to tailor the Hera models based on the UP attributes. An excerpt of the UP vocabulary is given in Figure 3.3.

We present the adaptation model when we show the different design steps. If the designer is not interested in adapting the system he can ignore the adaptation aspects in the proposed methodological steps. The adaptation presented here is a fine-grained adaptation. A coarse-grained adaptation is achieved by using group profiles, instead of UPs. In this approach users with similar characteristics are assigned a group profile. One of the advantages of coarse-level adaptation is the decrease in the system's workload, as the performed adaptation is relevant for several users.

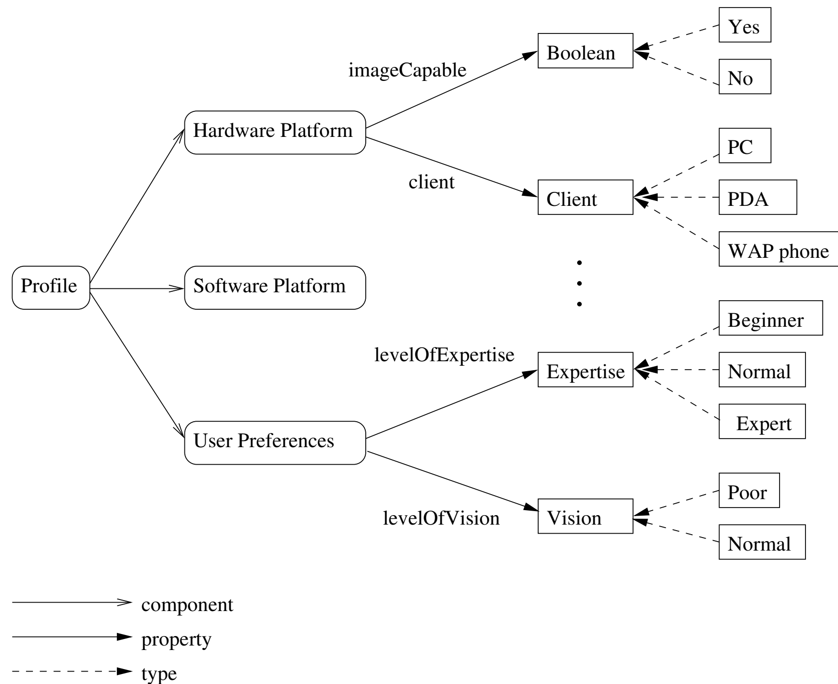

Figure 3.3: User/platform vocabulary.

3.3.1 Conceptual Design

The conceptual design specifies the input data in a uniform manner, independent from the input sources. The result of this activity is the *conceptual model* (CM). From a database point of view, the CM defines the schema for the data that needs to be presented. The CM serves also as the interface between the data collection phase and the presentation generation phase of the Hera methodology.

Figure 3.4 shows the CM vocabulary. It defines the following notions: *concept*, *concept attribute*, and *concept relationship*. A concept represents a certain entity in a particular application domain. Concept attributes and concept relationships refer to media types and other concepts, respectively, in order to describe the properties that characterize a concept. As CM vocabulary we did use the standard RDFS concepts with three extensions: one for modeling the *cardinality* of the concept relationships, one for representing the *inverse* of the concept relationships, and one for depicting the media types, the so-called *media vocabulary*. Similar to database modeling, many-to-many concept relationships are decomposed into two one-to-many concept relationships. In this way we have only two types of cardinalities: one-to-one and one-to-many.

Figure 3.5 shows the type hierarchy in the media vocabulary. In the same way as AMACONT [Fiala et al., 2003], we base our media vocabulary on a subset of the MPEG-7 standard [Martinez, 2003]. The basic media types are: *Text*, *Image*, *Audio*, and *Video*. The figure also shows the attributes of the media types, for example the *nrChars* of a text or the *width* and *height* of an image. For the refinement of the *Text* media types the XML

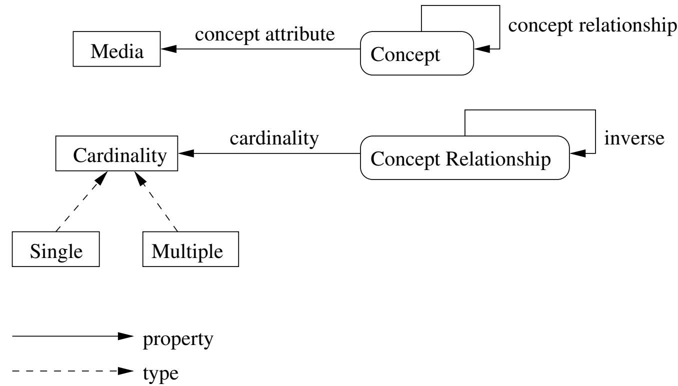

Figure 3.4: Conceptual model vocabulary.

Schema Datatypes (e.g., *String* or *Integer*) are used (not shown in the figure). One of the focus points of the Hera methodology is to reuse as much as possible the existing Web standards providing thus a higher degree of application interoperability.

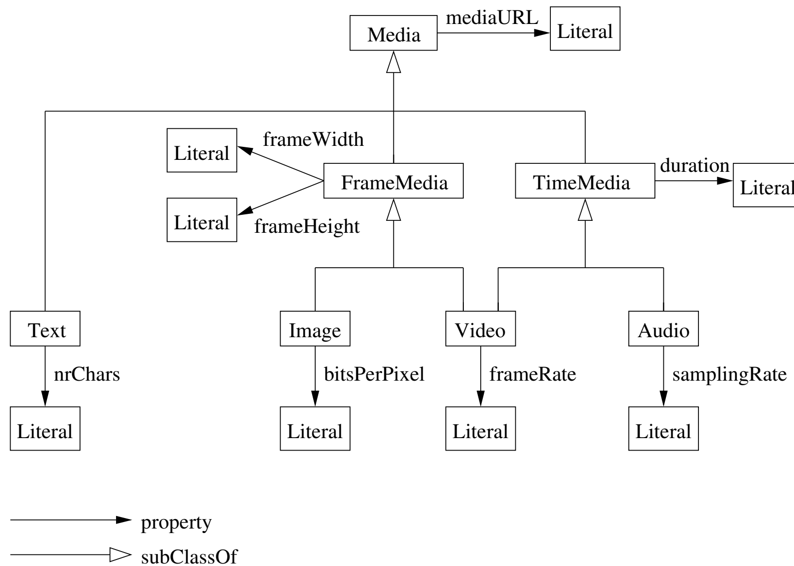

Figure 3.5: Media vocabulary.

Media adaptation selects the most appropriate media items for the technical system parameters provided by different network environments and client devices. Figure 3.6 shows a few media adaptation examples. For devices that are not able to display images (like certain WAP phones), the images are removed from the presentation. Based on display size, large strings and images are selected for PC, and small versions of the same strings and images are selected for PDA.

Figure 3.7 shows am excerpt of the CM for the running example. Concepts are represented as ovals and media types as rectangles. There are three concepts: *Technique*,

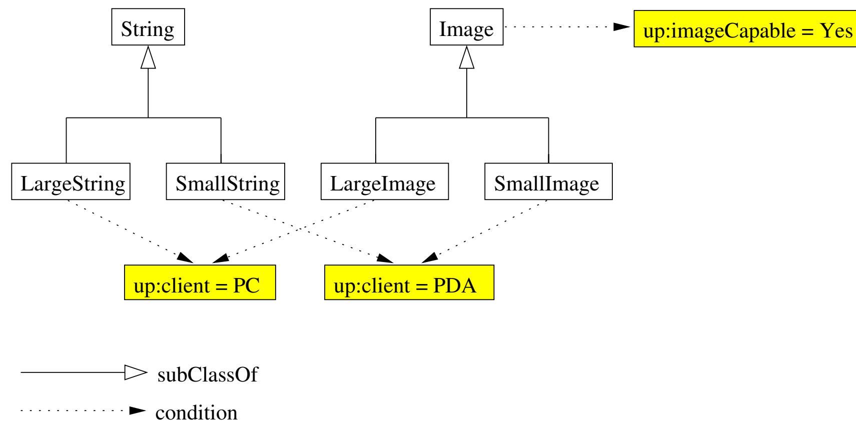

Figure 3.6: Media adaptation.

Artifact, and *Creator*. A *Creator* has two concept attributes attached to it, *cname*, for the creator's name, and *biography* for the creator's biography, both depicted by *String* items. A *Creator* is associated using the concept relationship *creates* to an *Artifact*. The cardinality of this concept relationship is one-to-many, i.e., one creator creates many artifacts. The inverse of the *creates* concept relationship is the *created_by* concept relationship. Note that both concept relationships and concept attributes are denoted as concept properties.

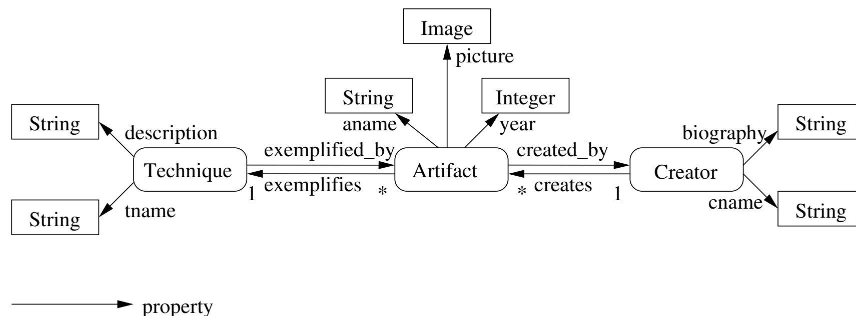

Figure 3.7: Conceptual model.

The conceptual model presented in Figure 3.7 depicting any creator, artifact, or technique can be refined to a specific artistic domain. Figure 3.8 shows the specialization (in a type hierarchy) of the previous conceptual model to the painting domain. Concepts are specialized by the *subClassOf* property and concept relationships are specialized by the *subPropertyOf* property. For example, the *Creator* is specialized as a *Painter* and the *creates* relationship is specialized as *paints*.

CM adaptation selects concepts or concepts attributes from the CM to be used in the presentation. Figure 3.9 shows an adaptation example in the conceptual model. In this example the *description* of the painting technique is removed from the whole presentation if the user is not an *Expert*. This is the so-called *context-independent adaptation*, i.e., adaptation that affects the entire presentation. An example of *context-dependent adaptation*,

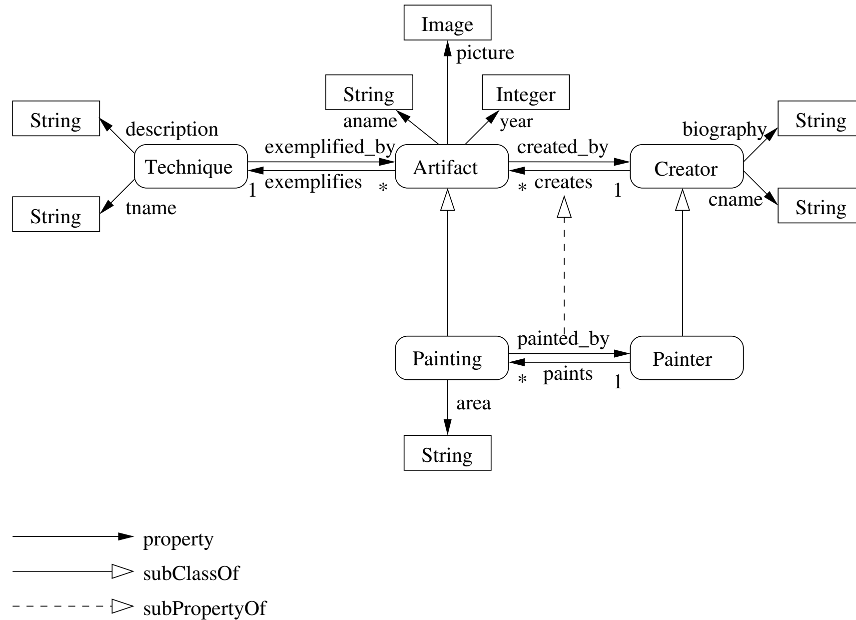

Figure 3.8: Specialization in the conceptual model.

i.e., adaptation that affects only a certain situation in a presentation, is provided in the next section.

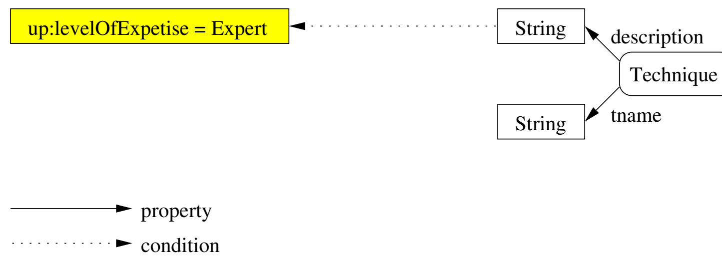

Figure 3.9: Adaptation in the conceptual model.

3.3.2 Application Design

The application design defines the navigational aspects of the presentation that is generated. A CM does not suffice to model a Web application [Rossi et al., 1999]: one needs to define the navigational view over the CM. The result of this activity is the *application model* (AM). From a database point of view, the AM is a view over the CM extended with navigation primitives.

Figure 3.10 shows the AM vocabulary. It defines the following notions: *slice*, *slice attribute*, and *slice relationship*. A slice [Isakowitz et al., 1998] is a meaningful presentation unit that fulfills a certain communication purpose. Slice attributes are used to refer to

media types. There are two types of slice relationships, *slice aggregation* and *slice navigation*. The first type of slice relationship facilitates the inclusion of a slice into another slice and the second type of slice relationship is used to define navigation between slices. An *empty slice*[1] is a slice that has its content defined at design-time. Such a slice has only one attribute that refers to a media type added at design-time. A *non-empty slice* has its content defined at run-time. In order to know from where the content is to be extracted at run-time slices have associated to them an *owner* concept from CM. The *owner* attribute for an empty slice can be any concept, as the slice content is defined at design-time.

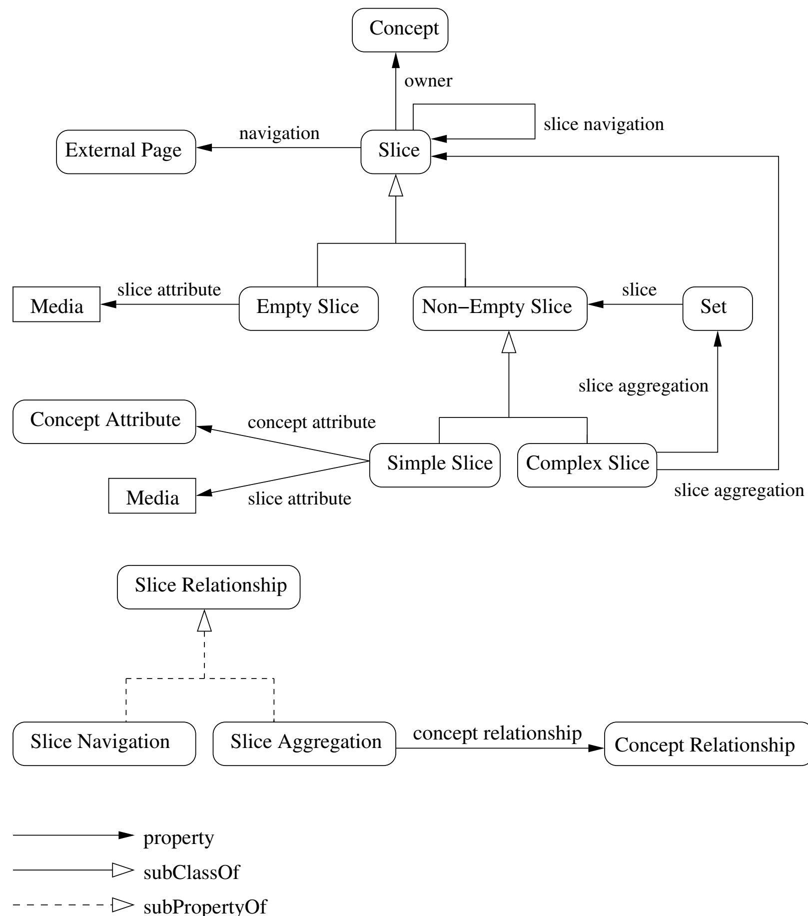

Figure 3.10: Application model vocabulary.

The definition of a non-empty slice is recursive: a slice can be a *simple slice* or can contain other slices[2]. A simple slice has only one slice attribute that refers to the same media

[1]Dealing with data-intensive applications, by 'empty' is meant that there is no content that will populate this type of slice at run-time.

[2]Due to their nested nature, slices are also called M-slices where 'M' stands for Matryoshka, the Russian doll [Diaz et al., 1997].

as the *concept attribute* of the owner concept from CM. A slice that aggregates other slices is called a *complex slice*. The recursion is defined by utilizing the *slice aggregation* relationship. The aggregation relationship between two slices that have two different owners needs to specify the *concept relationship* (or a relationship derived from the CM by relationship chaining) between the two owner concepts from the CM that made such an embedding possible. In case that the cardinality of this concept relationship is one-to-many the *Set* construct needs to be used. A *top-level slice* corresponds to a Web page. Using a slice navigation relationship, a slice (the anchor) can be linked to a top-level slice. Additionally a slice can be linked to an external Web page.

Figure 3.11 shows an excerpt of the AM for the running example. Slices are depicted (as their name suggests) by pizza-slice shapes. There are two slices, the main slice owned by *Technique* and the main slice owned by *Artifact*. We use the convention to denote the slice (long) name by *Slice.<concept name>.<slice short name>*, in order to distinguish them from concept names or slices with the same short name but owned by different concepts. The name of the slice owned by *Technique* is thus *Slice.Technique.main*. The slice *Slice.Technique.main* aggregates (by means of slice aggregation relationships) two simple slices and one complex slice. The simple slices *Slice.Technique.tname* and *Slice.Technique.description* are owned by *Technique*. The complex slice that aggregates *Slice.Artifact.picture* is owned by a different concept, i.e., *Artifact*. The aggregation relationship used for this embedding refers to the *exemplified_by* concept relationship between *Technique* and *Artifact*. As the cardinality of *exemplified_by* is one-to-many the *Set* construct is also inserted. In a similar manner the slice *Slice.Artifact.main* is defined. As *created_by* has cardinality many-to-one (inverse of *creates*), the *Set* construct is not used in this case. The slice navigation relationship connects the picture of an artifact *Slice.Artifact.picture* with the slice giving detailed information about that artifact *Slice.Artifact.main*.

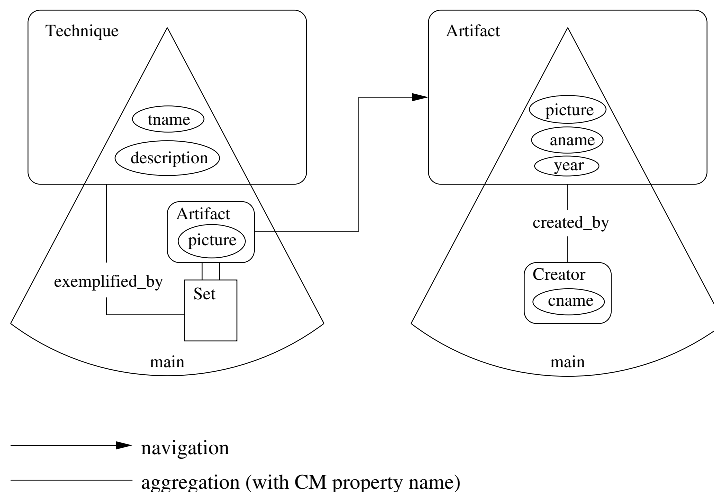

Figure 3.11: Application model.

The AM presented in Figure 3.11 depicting the main slices for techniques and ar-

tifacts can be refined to a specific artistic domain. Figure 3.12 shows the specialization (in a type hierarchy) of the previous AM to the painting domain. Slices are specialized by the *subClassOf* property. For example, the slice *Slice.Creator.main* is specialized by the slice *Slice.Painting.main*. *Slice.Painting.main* inherits all the slice relationships of *Slice.Technique.main* and adds three new slice relationships to it: two slice aggregations and one slice navigation. The aggregation relationships refer to the slice *Slice.Technique.area* and *Slice.Technique.tname*. The navigation relationship links backwards the *Slice.Technique.tname* with the *Slice.Technique.main*.

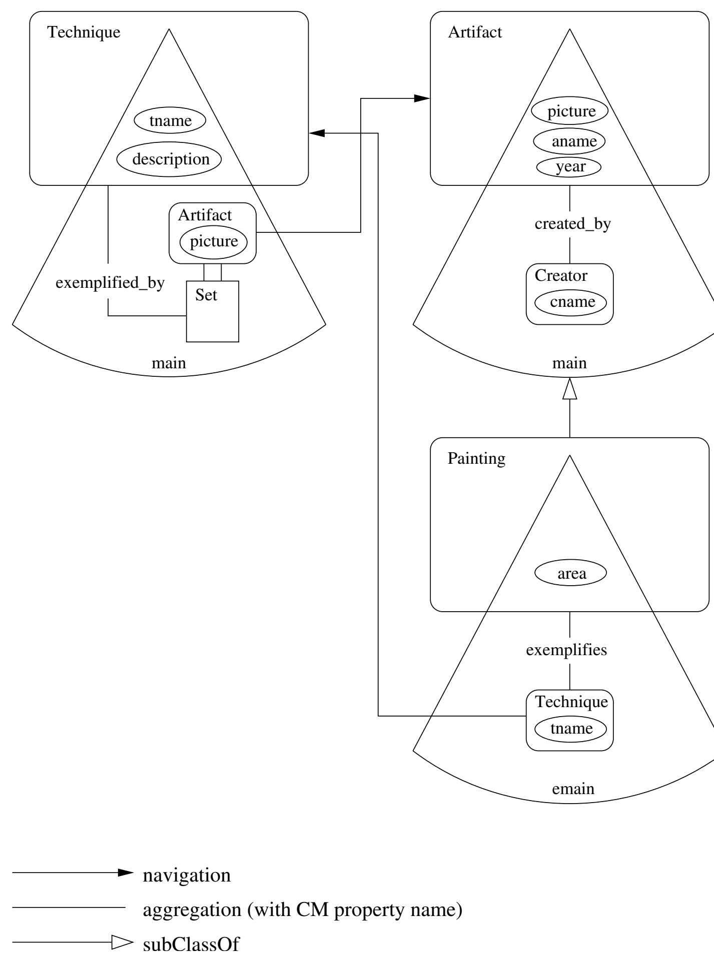

Figure 3.12: Specialization in the application model.

The AM adaptation [Frasincar and Houben, 2002] is based on two typical adaptation mechanisms: *conditional inclusion of fragments* (fragments are slices in our context) and *link hiding* [Brusilovsky, 2001] (links are slice navigation relationships in our context). A link is hidden when its destination slice has an invalid condition.

Figure 3.13 shows an adaptation example in the AM. In this example the *description* of the painting technique is removed from the main slice of this technique if the user is not an *Expert*. Later on in the presentation, the description of the painting technique can

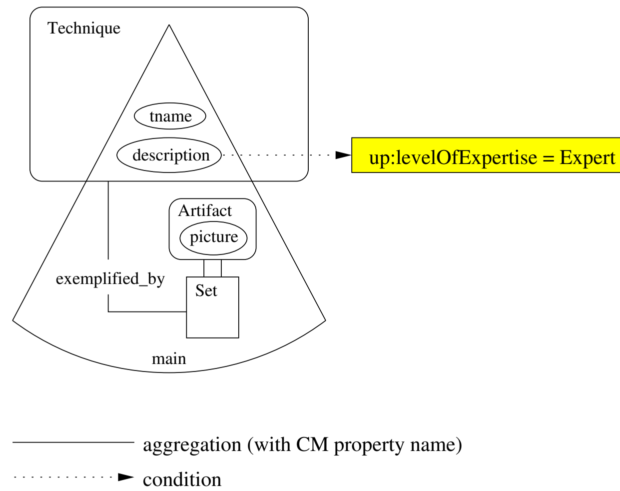

Figure 3.13: Adaptation in the application model.

appear also for users that are not *Expert*s (at that point in the presentation, the system can consider that the user is now ready to digest more advanced information). This is the so-called *context-dependent adaptation*, i.e., adaptation that affects only the current slice (by current slice is meant the top-level slice that contains the slice with the condition). Slices that have attached conditions outside the scope of a container slice have a *context-independent adaptation*, i.e., these slices will be removed from the whole presentation, no matter where they appear. This is similar to the *context-independent adaptation* for conceptual model adaptation showed in Figure 3.9. Note that the removal of a concept or concept attribute from a presentation has as its consequence the removal of all associated slices (i.e., slices for which the concept is an owner) and of the slice that refers to that concept attribute, respectively.

3.3.3 Presentation Design

The presentation design specifies the look-and-feel aspects of the presentation that is generated, independent from the implementation. The result of this activity is the *presentation model* (PM). It describes the layout and style information of the presentation. Both aspects are not to be neglected because they might have an immediate impact on the user choice for a certain application among applications offering similar functionality.

Figure 3.14 shows the PM vocabulary. It defines the following notions: *region*, *region attribute*, and *region relationship*. A region is an abstraction for a rectangular part of the display area where the content of a slice will be displayed. Each region is associated to a slice, the so-called region *owner*, from which the region content will be derived. The definition of region is very similar to that of a slice with a few simplifications and some additions. Region attributes are used to refer to media types. There are two types of region relationships, *region aggregation* and *region navigation*. The first type of region relationship facilitates the inclusion of a region into another region and the second type

of region relationship is used to define navigation between regions. The classification empty/non-empty does not apply for regions as regions get their content from the slice owner always at run-time.

The definition of regions is recursive: a region can be a simple region or can contain other regions. A *simple region* has only one region attribute that refers to the same media as the slice attribute of the corresponding simple slice from AM. Differently than for slices, one doesn't need to specify a corresponding concept attribute. A region that aggregates other regions is called a *complex region*. The recursion is defined by utilizing the region aggregation relationship. Another difference from slices is that for aggregation relationships there is no need to specify concept relationships. The *Set* construct, aggregation, and navigation relationships are copied for a region from the corresponding (by the owner relationship) slice. A *top-level region* corresponds to a Web page and is owned by a top-level slice.

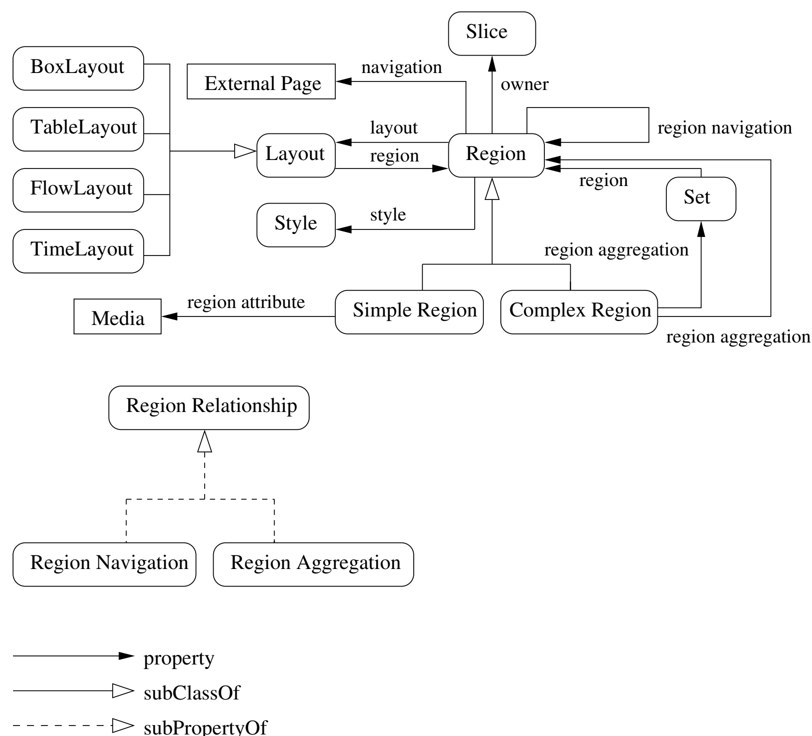

Figure 3.14: Presentation model vocabulary.

A region has a particular *layout manager* and *style* associated with it. There are four abstract layout managers: *BoxLayout*, *TableLayout*, *FlowLayout*, and *TimeLayout*. The layout managers describe the spatial/temporal arrangements of regions embedded into another region. The list of layout managers can be easily extended with other layouts like *BorderLayout*, *OverlayLayout*, *GuidedTourLayout*, etc.

[Frasincar et al., 2001] presents an alternative way of defining layouts by using qualitative and quantitative constraints for regions. These constraints are associated to region

relationships which are further classified as temporal, navigational, and spatial. Temporal relationships express the notion of time, navigational relationships represent (hyper)links, and spatial relationships define the spatial arrangements in presentations.

The layout managers were inspired from the abstract user interface (XML) representations from AMACONT [Fiala et al., 2004], UIML, and XIML [Souchon and Vanderdonckt, 2003]. These layout managers describe client-independent layouts that allow to abstract from the exact features of the browser's display. Note that because regions can be aggregated, layouts can also be aggregated (by means of regions), and thus one is able to build complex layouts.

The style information describes the colors, fonts, backgrounds to be used in a region, etc. Regions that do not have explicitly associated style information associated with them inherit the style of their container. In this way the designer is not forced to specify style information if that is not necessary.

The BoxLayout arranges the inner regions on one row or one column. Table 3.1 summarizes the possible attributes of the BoxLayout. The *height*, *width*, *border*, and *space* attributes have integer values that represent number of pixels.

Attribute	Meaning	Usage	Values
axis	orientation of the layout	required	"x" | "y"
rows	number of rows	optional	integer
columns	number of columns	optional	integer
height	height of the layout	optional	integer|percentage
width	width of the layout	optional	integer|percentage
border	size of the layout border	optional	integer
space	space between content and border	optional	integer

Table 3.1: BoxLayout attributes.

TableLayout arranges the inner regions in a table. Though it can be realized by nested *BoxLayout*s, we implemented it separately because SWISs often present dynamically retrieved sets of data in a tabular way. Table 3.2 summarizes the possible attributes of the TableLayout. Due to the dynamic nature of SWIS applications, the number of items in a complex region that uses the *Set* construct is not known at design-time. In such cases one should use only one of the dimensions: *rows* or *columns*. The missing dimension is automatically computed at run-time.

FlowLayout arranges the inner regions in the same way as words on a page: the first line is filled from left to right, then does the same for the lines below. Table 3.3 summarizes the possible attributes of the FlowLayout.

TimeLayout shows the inner regions in a time sequence that produces a slide show. Table 3.4 summarizes the possible attributes of the TimeLayout. The *duration* attribute has a float value that represents number of seconds. TimeLayout is used for platforms that support time sequences for presenting media items, e.g., Timed Interactive Multimedia

Attribute	Meaning	Usage	Values
rows	number of rows	optional	integer
columns	number of columns	optional	integer
height	height of the layout	optional	integer|percentage
width	width of the layout	optional	integer|percentage
border	size of the layout border	optional	integer
space	space between content and border	optional	integer

Table 3.2: TableLayout attributes.

Attribute	Meaning	Usage	Values
border	size of the layout border	optional	integer
space	space between content and border	optional	integer

Table 3.3: FlowLayout attributes.

Extensions for HTML (HTML+TIME) [Schmitz et al., 1998] and Synchronized Multimedia Integration Language (SMIL) [Ayars et al., 2005].

Attribute	Meaning	Usage	Values
duration	play time for a sequence element	optional	integer
repeat	number of times to repeat one sequence	optional	"indefinite"|integer

Table 3.4: TimeLayout attributes.

Table 3.5 summarizes the possible layout-related attributes for a region used inside a BoxLayout, TableLayout, or FlowLayout. These attributes describe how each referenced region has to be arranged in its surrounding layout. For example, the regions embedded in a layout form a sequence for which the order needs to be specified. For this purpose the *order* attribute is used. Note that for the TableLayout, the cell elements are counted from left to right and from top to bottom. The *sort* attribute specifies the sorting criteria for region instances. For example *alpha(Slice.Technique.tname,ascending)* specifies an alphabetical sorting in ascending order based on the name of artistic techniques. Besides the existing sorting functions like *alpha* and *num*, for alphabetical and numerical sorting, one can use its own sorting function (e.g., a multi-sort for data with different facets). If the sort criteria is not provided, the regions will be arranged in the order in which region content (data) is given by the data collection phase.

Even though most attributes are platform-independent, there are platform-dependent attributes in order to consider the specific card-based structure of WML presentations. The optional attribute *wml_visible* determines whether in a WML presentation a region should be shown on the same card. If not, it is put onto a separate card that is accessible

by an automatically generated hyperlink, the text of which is defined in *wml_description*. The *wml_description* attribute can refer to a constant string or one of the simple slices that give some of the content for a region. Note that this kind of content separation provides scalability by fragmenting the presentation according to the small displays of WAP phones.

Attribute	Meaning	Usage	Values
valign	vertical alignment	optional	“left”|“center”|“right”
halign	horizontal alignment	optional	“top”|“center”|“bottom”
ratio	space to be filled	optional	percentage
order	order in the sequence	optional	integer
sort	sorting criteria	optional	string
wml_visible	show on same card	optional	boolean
wml_description	anchor description	optional	string

Table 3.5: Layout-related region attributes inside BoxLayout/TableLayout.

Table 3.6 summarizes the possible layout-related attributes for a region used inside a FlowLayout. It is a subset of the previous set of attributes.

Attribute	Meaning	Usage	Values
order	order in the sequence	optional	integer
sort	sorting criteria	optional	string
wml_visible	show on same card	optional	boolean
wml_description	anchor description	optional	string

Table 3.6: Layout-related region attributes inside FlowLayout.

Table 3.7 summarizes the possible attributes for a region used inside a TimeLayout. The *begin*, *duration*, and *end* attributes have float values that represent the number of seconds.

Attribute	Meaning	Usage	Values
begin	(absolute) start time	optional	float
duration	play time	optional	float
end	(absolute) end time	optional	float

Table 3.7: Layout-related region attributes inside TimeLayout.

Table 3.8 presents some of the possible style attributes. These attributes refer to the font characteristics (e.g., size, color), background, link colors, etc. The definition of these attributes is inspired from Cascading Style Sheets (CSS) [Bos et al., 2004].

Attribute	Meaning	Usage	Values
font-family	the family of a font	optional	“times”|“helvetica”| ...
font-style	the style of a font	optional	“normal”|“italic”
font-size	the size of a font	optional	“small”|“medium”|“large”
font-color	the color of a font	optional	“red”|“green”|...
font-weight	the weight of a font	optional	“normal”|“bold”|...
background-color	the color of the background	optional	“red”|“green”|...
link-color	the color of a not-visited link	optional	“red”|“green”|...
visited-color	the color of a visited link	optional	“red”|“green”|...
...			

Table 3.8: Style attributes.

The layout managers need to be instantiated in order to be used in the PM. The layout manager instances are used for complex regions. Also when referencing a region (or set of regions) one needs to define values for the layout-related region attributes corresponding to the layout associated to the container region.

Figure 3.15 shows an excerpt of the PM for the running example. Regions are depicted as rectangles. There are two top-level regions: *RegionFullT* and *RegionFullA*. *RegionFullT* and *RegionFullA* are owned by *Slice.Technique.main* and *Slice.Artifact.main*, respectively. We use the convention to denote the region (long) name by *Region.<Slice full name>.<Region short name>*. The short name of a region can be omitted from its full name, if the full name unambiguously identifies the region. The full name of *RegionFullT* is *Region.Slice.Technique.main.RegionFullT*. As the full names are quite long in the rest of the explanation it is used the short name of regions when these short names are available.

The region *RegionFullT* aggregates (by means of slice aggregation relationships) three regions: one contains the technique name, one contains the technique description and, one contains the set of pictures that exemplify a painting technique. As simple regions, the first two regions do not need a layout. The third region, a complex region, has a *TableLayout* specified for arranging the set of pictures. All three regions are arranged using a *BoxLayout* specified in the *RegionFullT*. The style information is given by the *DefaultStyle*. As can be seen from the figure the inner regions do not have the style information explicitly defined which means that they inherit the style information from the container region. In a similar manner is defined the region *RegionFullA*. The region navigation relationship connects *RegionBottomA* with *RegionFullA*.

Figure 3.16 shows some of the layout attributes and layout-related region attributes for our running example. *RegionFullT* has a BoxLayout with two attributes defined: *axis* with value *y* which indicates that this layout has a vertical arrangement and *width* with value *100%* which means that this layout will completely fill the width of its container. As *RegionFullT* is a top-level region, the container is the user's display. In *BoxLayout1* there are three regions embedded in the order specified by the *order* attribute. All three regions have the *halign* layout-related attribute defined in order to specify that their hor-

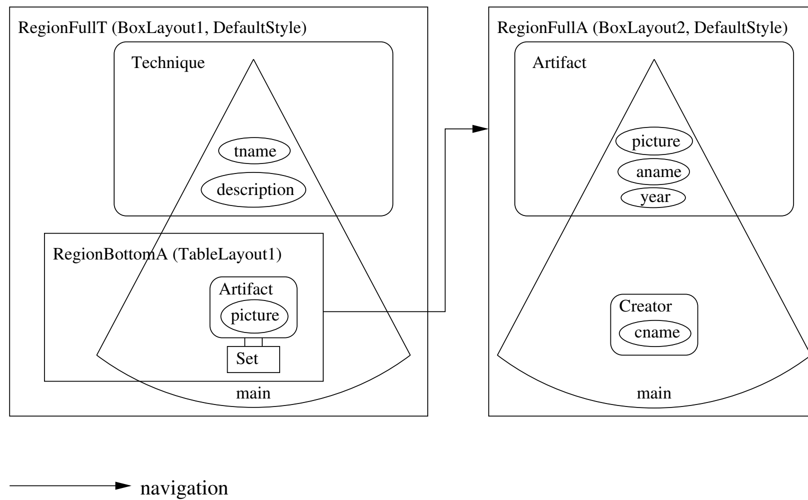

Figure 3.15: Presentation model.

izontal alignment will be centered. The third layout, *RegionBottomA* has two attributes defined: *cols* with value *3* which indicates that this layout has three columns and *width* with value *100%* which means that this layout will completely fill the width of its container. The container is in this case *RegionFullT*. *RegionBottomA* contains pictures for which the horizontal alignment is centered.

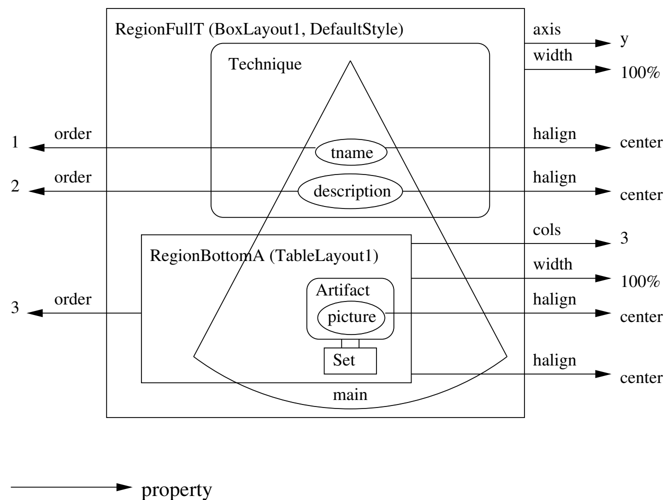

Figure 3.16: Layout and layout-related region attributes.

The PM presented in Figure 3.15 depicting the main regions for techniques and artifacts can be refined to a specific artistic domain. Figure 3.17 shows the specialization (in a type hierarchy) of the previous PM to the painting domain.

Regions are specialized by the *subClassOf* property. For example, the region *RegionFullA* is specialized by the region *RegionFullP*. *RegionFullP* inherits all the region relation-

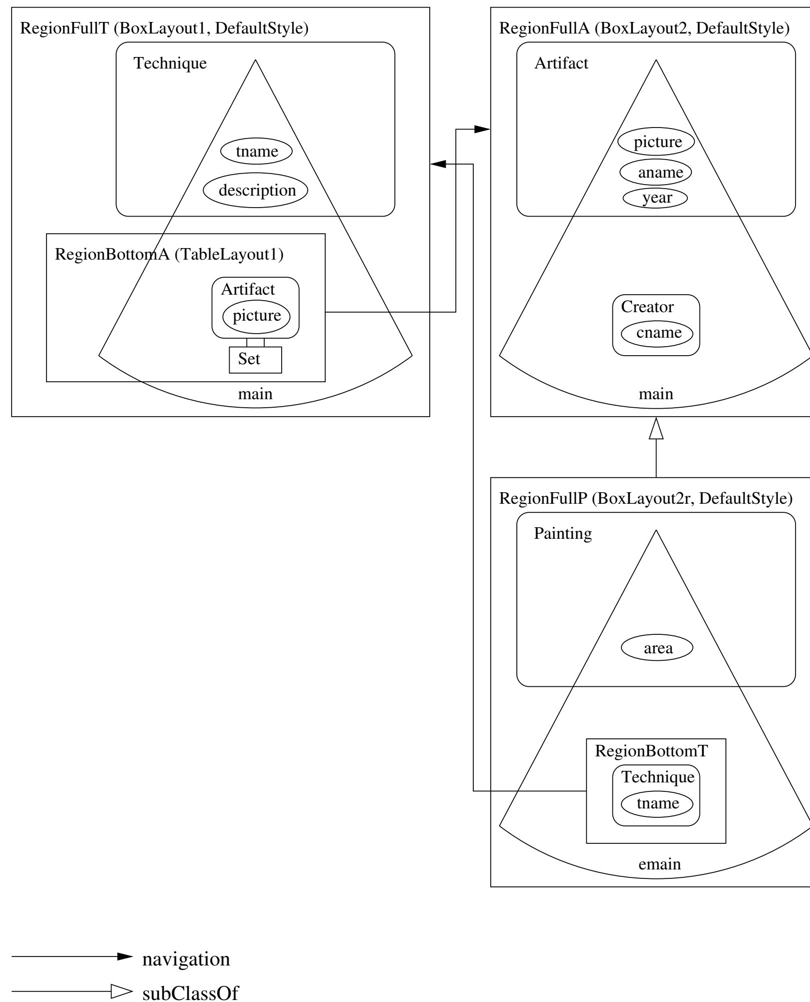

Figure 3.17: Specialization in the presentation model.

ships of *RegionFullA* and adds three new region relationships to it: two region aggregations and one region navigation. The aggregation relationships refer to the regions *Region.Slice.Painting.area* and *RegionBottomT*. As *RegionFullP* contains more regions than *RegionFullA*, the *BoxLayout2* is replaced with *BoxLayout2r* which among other things specifies in which order the added regions are placed. The navigation relationship links backwards the *RegionBottomT* with *RegionFullT*.

PM adaptation selects layouts or styles from PM to be used in the presentation. Figure 3.18 shows two adaptation examples in PM. In one example, depending on the size of the screen, the *RegionBottomA* uses a *BoxLayout* for PDA and a *TableLayout* for PC. The small screen size of the PDA requires a vertical arrangement of the data. In the other example the *DefaultStyle* uses medium fonts for a user with a average level of vision and large fonts for a user with a low level of vision. Other possible adaptation examples are: increasing the font of links for users with limited manual dexterity, eliminate colors for

color-blind users, etc.

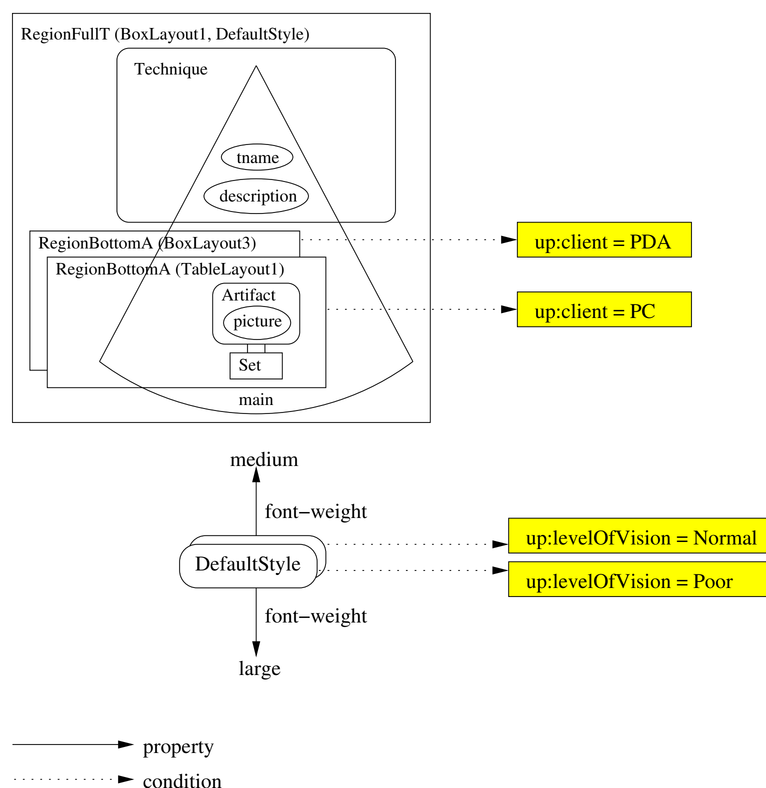

Figure 3.18: Adaptation in the presentation model.

3.3.4 Implementation

The implementation of the static variant of the Hera presentation generation phase is based on several data transformations specified by XSLT [Kay, 2005b] stylesheets. These transformations operate on the RDF/XML [Beckett, 2004] serialization of the RDF models. The XSLT processor used for interpreting XSLT stylesheets is Saxon [Kay, 2005a]. Figure 3.19 shows the transformation steps for the static variant of the Hera presentation generation phase. Each transformation step has a label associated with it. Some of these transformations have substeps which are labeled using a second digit notation.

In Figure 3.19 there are two types of dashed arrows: “is used by” to express that an RDFS model is used by another RDFS model and “has instance” to denote that an RDFS model has as instance an RDF model. A model vocabulary, a model, a model instance, and the generated presentations are depicted by rectangles. The transformation specifications are represented by ovals.

There are three types of model/transformation specifications: application-independent, application-dependent, and query-dependent. The application-independent specifications

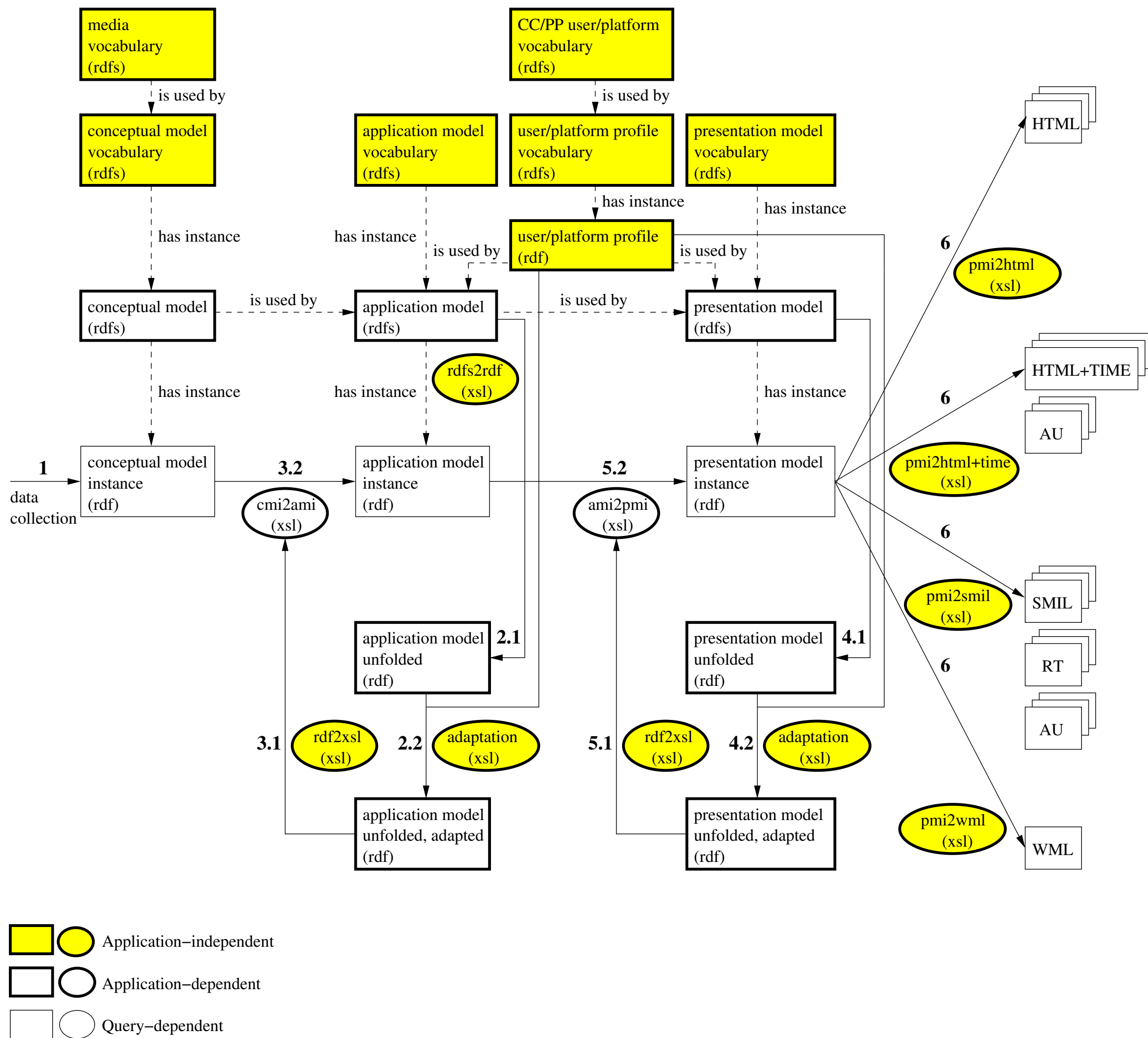

Figure 3.19: Presentation generation using XSLT.

do not refer to SWIS models (CM, AM, and PM), the application-dependent specifications refer to SWIS models, and query-dependent specifications refer to the SWIS models and the retrieved data (e.g., model instances). One can note that the query-dependent transformations are also application-dependent transformations. Transformations that are application-independent are also called generic transformations. Transformations that are application-dependent are also called specific transformations.

The input to the presentation generation phase is the conceptual model instance (CMI), i.e., the data retrieved in response to a user query. This data is produced in the *data collection* phase from a given set of input sources. This is step 1 in the figure and is not described here. More information on step 1 can be found in [Vdovjak et al., 2003]. At the current moment CM and media adaptation are carried on in the AM adaptation. Future implementations will separate the CM and media adaptation from the AM adaptation.

Step 2, the *AM generation*, builds an adapted AM template. This step contains two substeps: the *AM unfolding* and the *AM adaptation*.

Step 2.1, the *AM unfolding*, generates the AM template. The AM template represents the structure of an AM instance (RDF) based on the AM schema (RDFS). Such a template will ease the specification of an XSLT stylesheet used to convert a CM instance (CMI) to an AM instance (AMI). By unfolding the AM we mean repeating the process of adding properties inside the subject classes until slice references or media items are reached. In this way one obtains an AM template which will be filled later on with appropriate instances.

Step 2.2, the *AM adaptation*, executes the adaptation specifications on the AM template. The transformation stylesheet of this step has two inputs: the AM template and the UP. The UP attributes are replaced in the conditions by their corresponding values. The slices that have the conditions not valid are discarded and the hyperlinks pointing to these slices are disabled.

Step 3, the *AMI generation*, instantiates the AM with the retrieved data. This step is composed of two substeps: the *AMI transformation generation* and the *AMI creation*.

Step 3.1, the *AMI transformation generation*, builds the transformation stylesheet that will convert a CMI to an AMI. This step uses an XSLT stylesheet that will generate another XSLT stylesheet. One should note that an XSLT stylesheet is a valid XML file that can be produced by another XSLT stylesheet. This technique was also successfully used in the previous version of the implementation which was XML-based [Frasincar and Houben, 2001]. This transformation is based on the *owner* of a slice and the *concept attribute* of a simple slice. The following name convention is used: a slice instance name (e.g., *Slice.Painting.main_ID1*) is obtained from the slice name (e.g., *Slice.Painting.main*) concatenated with the suffix (e.g., *ID1*) of the associated concept instance identifier (e.g., *Painting_ID1*). The implemented algorithm is straightforward: instantiate all slices for all the corresponding retrieved concept instances and each time a slice is referenced add its identifier based on the above name convention.

The transformation used in this phase is a generic one, but the output that it produces is used for a specific transformation (the next step).

Step 3.2, the *AMI creation*, converts the CMI to an AMI. The XSLT stylesheet obtained in the previous substep is applied to the CMI to yield an AMI. As opposed to the previous transformations, this stylesheet will operate for inputs and outputs that are both query-dependent. For each query, Hera will dynamically instantiate the AM with the query result, i.e., a CMI.

The PM-related transformation steps (steps 4 and 5) are realized in a similar manner as the AM-related transformation steps (steps 2 and 3).

Step 4, the *PM generation*, builds a PM template. This step contains two substeps: the *PM unfolding* and the *PM adaptation*.

Step 4.1, the *PM unfolding*, generates the PM template. The PM template represents the structure of a PM instance (RDF) based on the PM schema (RDFS). Such a template will ease the specification of an XSLT stylesheet used to convert an AM instance (AMI) to a PM instance (PMI). By unfolding the PM we mean repeating the process of adding properties inside the subject classes until slice references or media items are reached. In this

way, one obtains a PM template which will be filled later on with appropriate instances.

Step 4.2, the *PM adaptation*, executes the adaptation specifications on the PM template. The transformation stylesheet of this step has two inputs: the PM template and the UP. The UP attributes are replaced in the conditions by their corresponding values. The layouts and styles that have the conditions not valid are discarded.

Step 5, the *PMI generation*, instantiates the PM with data from the AMI. This step is composed of two substeps: the *PMI transformation generation* and the *PMI generation*.

Step 5.1, the *PMI transformation generation*, builds the transformation stylesheet that will convert an AMI to a PMI. As in step 3.1, an XSLT stylesheet that will generate another XSLT stylesheet is used. This transformation is based on the *owner* of a region and the fact that simple regions are associated to simple slices. The following name convention is used: a region instance name (e.g., *Region.Slice.Painting.main.RegionFullA_ID1*) is obtained from the region name (e.g., *Region.Slice.Painting.main.RegionFullA*) concatenated with the suffix (e.g., *ID1*) of the associated slice instance identifier (e.g., *Slice.Painting.main_ID1*). The implemented algorithm is straightforward: instantiate all regions for all the corresponding slice instances and each time a region is referenced add its identifier based on the above name convention.

Step 5.2, the *PMI creation*, converts the AMI to a PMI. The XSLT stylesheet obtained in the previous substep is applied to the AMI to yield a PMI. As opposed to the previous transformations, this stylesheet will operate for inputs and outputs that are both query-dependent.

Step 6, the *presentation data generation*, transforms the PMI into code specific for the user's browser. Note that a set of Web pages is generated at-a-time. Some of supported formats are: HTML, HTML+TIME, WML, and SMIL. For each type of serialization a specific stylesheet is used. The stylesheets used for the HTML, HTML+TIME, and SMIL use the ability of XSLT 2.0 [Kay, 2005b] to generate multiple outputs (this feature is not supported in XSLT 1.0 [Clark, 1999]). In order to generate multiple outputs the XSLT 2.0 *result-document()* function was used.

For HTML(+TIME), BorderLayout and TableLayout are implemented using tables. An HTML presentation is composed from the *index.html* document (starting point of the presentation) and a set of HTML pages each corresponding to a top-level slice.

The *FlowLayout* is supported by any HTML browser (the content of a table cell is automatically wrapped if it doesn't fit one line). *TimeLayout* is supported only by HTML+TIME and SMIL browsers.

For WML, there is only one layout supported, i.e., the *BorderLayout* with a vertical alignment. Because lists are not available in WML, they are implemented as simple sequences of items without any visual cues. To each top-level region corresponds a WML *card*. A WML presentation is composed from a single WML document, *a deck* that contains a set of cards. The first card is the starting point of the presentation.

For SMIL, there is an explicit part to describe the layout of a document. As tables/flow are not supported in SMIL, one needs always to fully define the layout information for *BoxLayout*, *TableLayout*, and *FlowLayout*. The *TimeLayout* was defined using the *seq* container for regions. Hera regions are implemented as SMIL regions. A SMIL presentation

is composed from a main SMIL document (starting point of the presentation), a set of SMIL documents each corresponding to a top-level region, a set of RealText (RT) clips, one per each text media, and a set of audio clips (AU), one per each audio media.

3.4 Presentation Generation (Dynamic)

Recently the Hera methodology has been extended in order to accommodate more complex forms of user interaction in addition to simple link-following, e.g., interaction by means of forms in which the user can enter data [Houben et al., 2004]. In this way the user can better personalize the SWIS according to his needs, specially regarding the dynamics within a browsing session. Figure 3.20 shows the "loop" with which we extended the presentation generation to support this additional dynamics and to allow the user to influence the generation of the Web presentation. Note that in response to a user query only one page is generated at-a-time instead of the full Web presentation as is the case for the static variant of the presentation generation phase. Generating one-page-at-a-time allows the system to consider the user input before generating the next Web page. The request contains the (owner) concept instance identifier and the slice type of the next slice to be generated (i.e., the one corresponding to the next Web page).

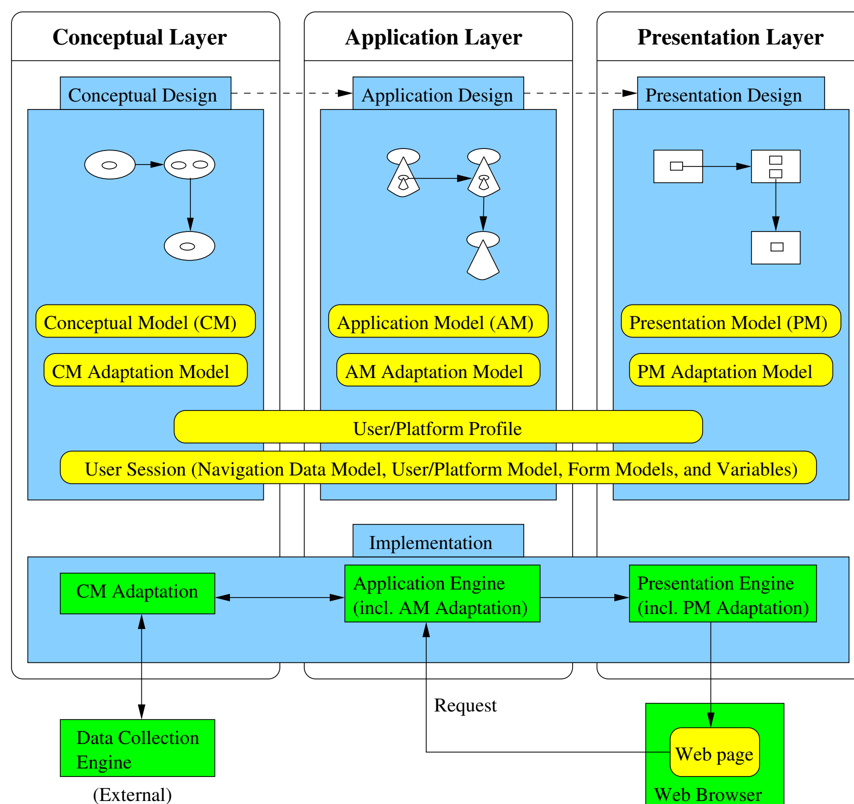

Figure 3.20: Presentation generation phase (dynamic).

In order to illustrate the dynamic version of the presentation generation the running example is extended such that it allows the visitor to buy posters of the paintings in the museum. For simplicity we didn't model explicitly the posters, assuming a one-to-one correspondence with the depicted painting. Also, after buying a certain painting, the user will not be presented with the same painting again.

In addition to the data from CM, AM, and PM, interaction requires a support for creating, storing, and accessing data that emerges while the user interacts with the system. This support is provided by means of the user session (US). US is composed of the *navigation data model*, *user/platform model*, *form models*, and *variables*.

The purpose of the navigation data model (NDM) is to complement the CM with a number of auxiliary concepts that do not necessarily exist in the CM (although this is the decision of the designer in concrete applications) and which can be used in the AM when defining the behavior of the application and its navigation structure.

The user/platform model (UM) stores user preferences and device capabilities that change during user browsing (e.g., network connection speed, user knowledge on some of the displayed topics, etc.). In Section 3.3 the UP was defined. The UP-based adaptation is done at the beginning of the user browsing session in order to adapt the CM, AM, and PM. In a similar way the UM is used to adapt the CM, AM, and PM. Differently than for UP, the UM-based adaptation is done before each Web page is generated.

The form models (FM) describe the data that is entered by the user by means of forms. Each form has a so-called form model associated with it. The data input by the user in a form populates the associated form model. Similar to XForms [Dubinko et al., 2003], a form separates presentation from content. FM describes the form content. The presentation-related issues of forms are given in the AM.

The session variables are the concept instance identifier, i.e., *instanceid*, and the slice type, i.e., *slicetype*, of the previous slice (the one from which a request originated), and a number of variables to store temporary data created during a user browsing session (e.g., for storing the URIs of newly created resources).

We remark that from the system perspective the concepts in the NDM can be divided into two groups. The first group essentially represents views over the concepts from the CM, the second group corresponds to a locally maintained repository. A concept from the first group can be instantiated only with a subset of instances of a concept existing in the CM, without the possibility to change the actual content of the data. A concept from the second group is populated with instances based on the user's interaction, i.e., the data is created, updated, and potentially deleted on-the-fly. The AM can refer to the concepts from NDM as if they were representing “real” data concepts.

The NDM of our example is depicted in Figure 3.21; it consists of the following concepts: *SelectedPainting*, *Order*, and *Trolley*. The *SelectedPainting* concept is a subclass of the *Painting* concept from the CM. It represents those paintings which the user selected in a selection form. The *Order* concept models a single ordered item consisting of a selected painting (the property *includes*) and the *quantity* represented by an Integer. The *Trolley* concept represents a shopping cart containing a set of orders linked by the property *contains*.

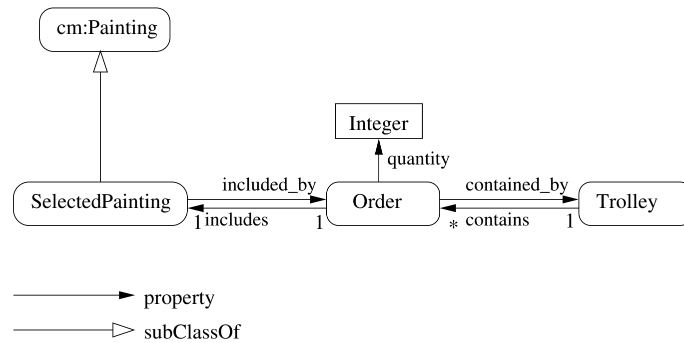

Figure 3.21: Navigation data model.

In the example the *SelectedPainting* concept belongs to the group of view concepts whereas both the *Order* and the *Trolley* are updatable concepts with the values determined at run-time. This is reflected also in the navigational data model instance (NDMI) depicted in Figure 3.22 that results from the user's desire to buy 1 poster of the selected painting. The instance *Painting*1 comes from the CM, i.e., it is not (re)created: what is created however, is the *type* property associating it with the *SelectedPainting* concept. Both instances *Order*1 and *Trolley*1 are created during the user's interaction; they, as well as their properties, are depicted in bold in Figure 3.22.

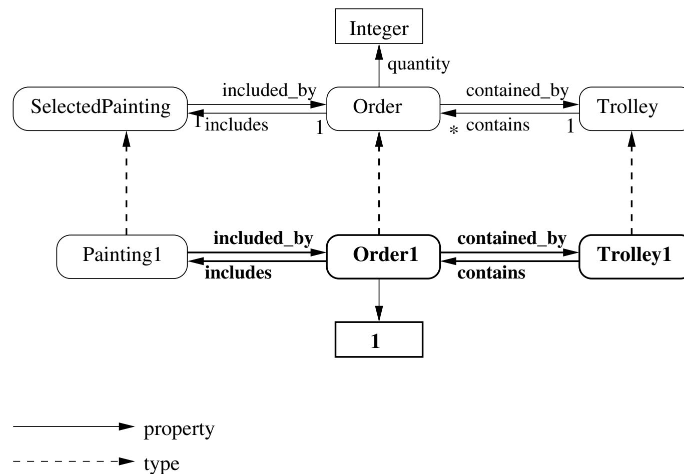

Figure 3.22: Navigation data model instance.

The application model vocabulary from Figure 3.10 was extended in order to support forms. Figure 3.23 shows these extensions, inspired by the XForms standard. Similar to XForms, a form separates presentation from content. The presentation-related issues of forms are associated to the AM. In AM, a *form* is a particular type of slice which has controls associated with it. Some of the supported form controls (as in XForms) are: *Select1* (*S1*), selects one instance from a set; *SelectN* (*SN*), selects several instances from a set; *Input (I)*, accepts one line of input text, etc.

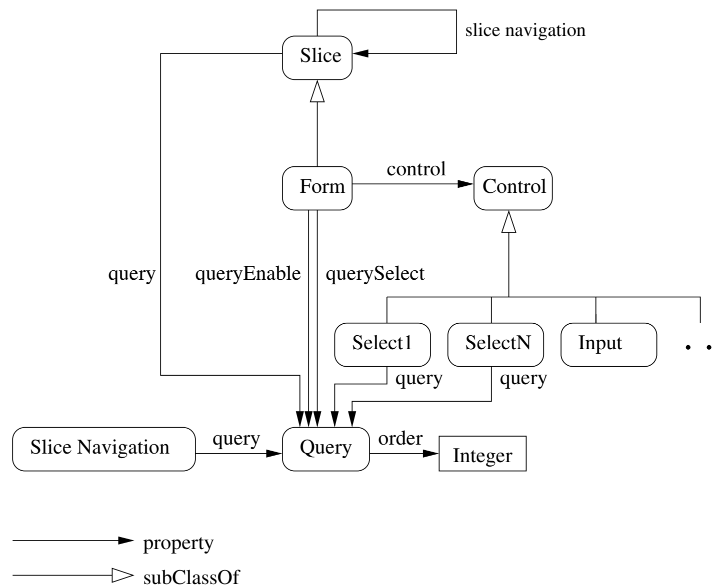

Figure 3.23: Extended application model vocabulary.

The dynamics of the application is given by a set of AM queries used for selection, deleting, or updating of data. These queries can be attached to:

- *slices*, to express user-independent updates (e.g., creation of a trolley),
- *form controls*, to get values for these controls (e.g., select all names of paintings that are not in the trolley),
- *forms*, (1) to enable/disable a form (e.g., if the user has already added all paintings to his trolley, there is no painting left to be offered to the user for the next selection, and therefore the selection form is disabled) or (2) to select the concept instance for the next slice (e.g., after selecting a painting, the main slice of the selected painting is presented),
- *slice navigation*, to express user-dependent updates (e.g., create order and add it to the trolley).

By a query that enables/disables a form it is actually meant a condition that uses some query results for enabling/disabling a form. The identification of the query with the condition is done because the condition usually is a very simple one (in most of the encountered cases it is a comparison of the query result with '0'). An element from AM can have attached a single query or a sequence of queries. The order in which the sequence queries will be executed is given by the *order* attribute.

The content of the form is based on a form model (FM), i.e., the schema of the data associated with a certain form. The data of the form that populates (at run-time, based on user actions) the FM is the so-called form model instance (FMI). The mappings (bindings)

of the data provided by the form controls to the form model instance is outside the scope of this description as this is done by an external XForms processor. Figure 3.24 shows an example of a form model and its instance.

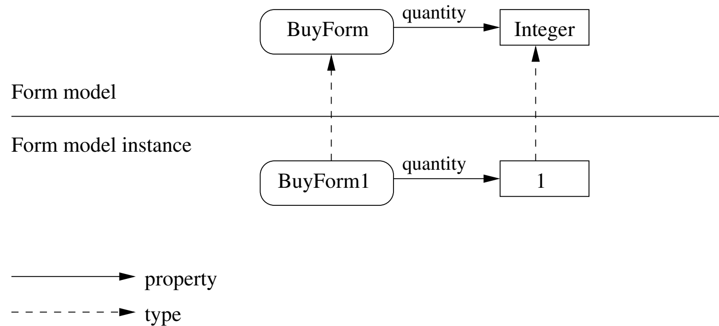

Figure 3.24: Form model and form model instance.

Figure 3.25 shows two form slices that can be embedded in an AM. The short names of the forms are *SelectForm* and *DeleteForm* and the long names are *Slice.Painting.SelectForm* and *Slice.Trolley.DeleteForm*, respectively. The owner of the *SelectForm* is *Painting* and the owner of the *DeleteForm* is *Trolley*. Two queries are used to enable/disable the forms: *QEnableSF* and *QEnableDF*. Both forms have one control field defined *S1* (selects one instance from a set). The values from which the user makes one selection are given by the queries *QSelectSFPn* and *QSelectDFPn*. The first form has *QSelectP* a query that selects a painting instance identifier based on the user's choice. The second form has a slice navigation relationship with an update query defined, i.e., *QDeleteO*.

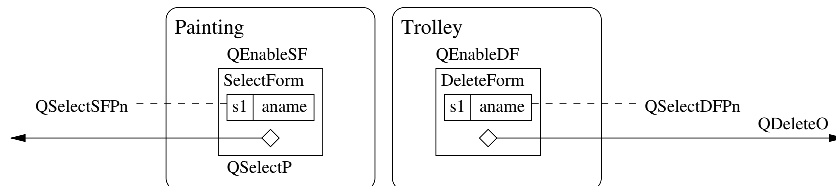

Figure 3.25: Form in application model.

Figure 3.26 shows the application model extended with forms. The main slice of a painting depicts information related to the painting. It also contains the *BuyForm*, a form that allows the user to make an order by specifying the quantity of desired posters for the presented painting. In order not to produce too much visual clutter, we do not show in the figure the concept owner of the form (this is the same as the owner of the destination slice when one navigates from that form). The main slice of the trolley displays the orders contained in the trolley. Note that when the user makes an order, this order is immediately added to the trolley. In addition the main slice of the trolley has two other forms *SelectForm* and *DeleteForm*. *SelectForm* is used to select paintings by their name, paintings which do not have posters in the trolley. *DeleteForm* is used to delete orders from the trolley.

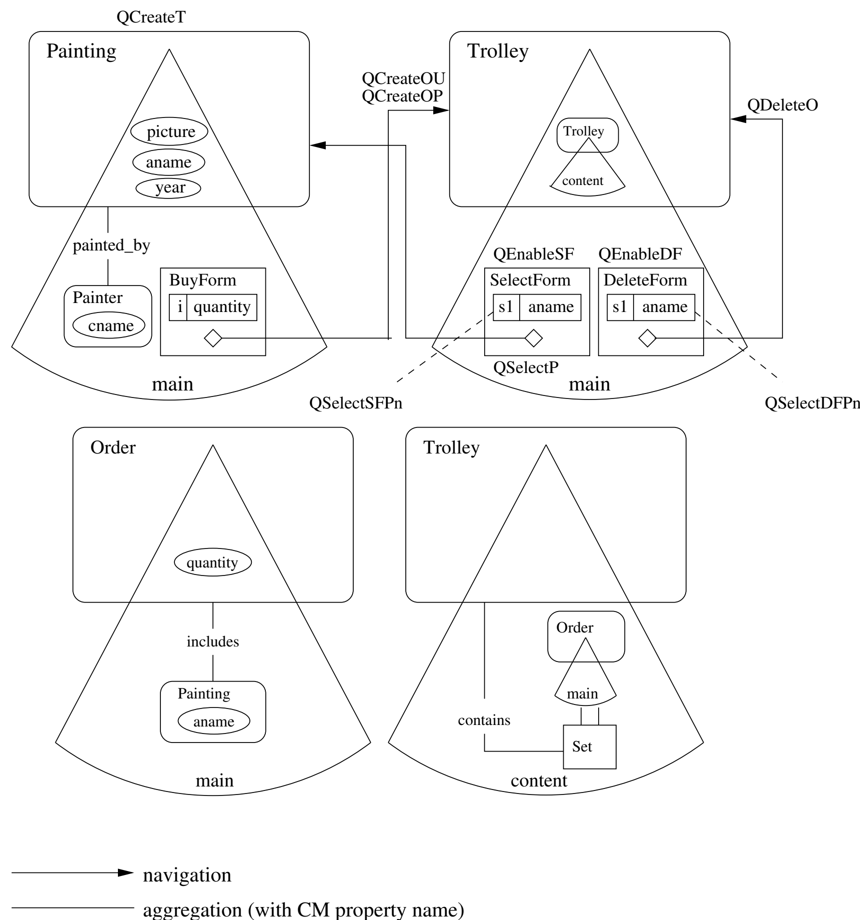

Figure 3.26: Extended application model.

Because models are represented in RDF(S), the AM queries are described using an RDF query language. As an RDF query language it was chosen SeRQL [Aduna, BV, 2005], one of the most expressive RDF query languages that supports not only the selection of RDF data but also the creation of new RDF data. In the rest of this section several queries are presented in their SeRQL syntax. Due to the fact that SeRQL doesn't support nested queries some queries are expressed in RQL [Karvounarakis et al., 2002]. In the rest of this section the queries from Figure 3.26 are presented.

Figure 3.27 shows *QCreateT* a query attached to the main slice of a painting. It is used to create a trolley for the user. The SeRQL was extended with the `new()` function that is able to create a URI (identifier) unique in the application for a new resource. The newly created URI is stored in the user session variable `trolleyid`.

```
CONSTRUCT {new()}<rdf:type><ndm:Trolley>
```

Figure 3.27: QCreateT (create trolley).

QCreateOU and *QCreateOP* are a sequence of queries attached to the slice navigation from *BuyForm* to the main slice of the trolley. Figure 3.28 depicts *QCreateOU*, a query that creates a new order. The newly created URI is stored in the user session variable *orderid*.

```
CONSTRUCT {new()}<rdf:type><ndm:Order>
```

Figure 3.28: QCreateOU (create order).

Figure 3.29 shows *QCreateOP*, a query that fills the order properties and adds the order to the trolley. Note that the order is captured in NDM, the owner concept instance identifier of the current slice and the newly generated order identifier are user session variables, and the user input (the poster's quantity) is captured in *BuyForm1*, the form model instance of the form *BuyForm*.

```
CONSTRUCT
     {x}<ndm:contains>{y},
     {y}<ndm:contained_by>{x},
     {y}<ndm:includes>{z},
     {z}<ndm:included_by>{y},
     {y}<ndm:quantity>{v}
FROM
     {session}<var:trolleyid>{x},
     {session}<var:instanceid}{z},
     {session}<var:orderid>{y},
     {BuyForm1}<bf:quantity>{v}
```

Figure 3.29: QCreateOP (add order to trolley).

Figure 3.30 shows *QEnableSF*, a query attached to the *SelectForm* form in order to enable/disable this form. If all paintings have orders associated with them, the *SelectForm* is disabled, as there are no paintings left for user selection. SeRQL was extended with aggregation functions like the *count()* function.

```
(SELECT count(x)
 FROM {x}<rdf:type><cm:Painting>
 WHERE NOT x IN SELECT y
                FROM {session}<var:trolleyid>{v},
                     {v}<ndm:contains>{w},
                     {w}<ndm:includes>{y}) > 0
```

Figure 3.30: QEnableSF (condition that enables/disables SelectForm).

Figure 3.31 shows *QSelectSFPn*, a query attached to the control of the form *SelectForm* in the main slice of trolley. Note that *QSelectSFPn* is a nested query: first the paintings included in the order are computed and the result is subtracted from the set of all the paintings. The query returns the name of the paintings that are not in the trolley.

```
SELECT xname
FROM {x}<rdf:type><cm:Painting>,
     {x}<cm:aname>{xname}
WHERE NOT x IN SELECT y
               FROM {session}<var:trolleyid>{v},
                    {v}<ndm:contains>{w},
                    {w}<ndm:includes>{y}
```

Figure 3.31: QSelectSFPn (select paintings (names) that are not in the trolley).

Figure 3.32 shows *QSelectP*, a query attached to the *SelectForm* in order to select the concept instance that owns the next slice to be presented (i.e., the main slice of a painting). In the future we would like to exploit this selection feature (based on queries) at a more general level, i.e., in the navigation between any two slices and not just between forms (form slices) and slices. In this way the restriction that slice navigation relationships connect slices that have the same owner will be eliminated. Nevertheless one should ensure that only one instance of the destination slice is created.

```
SELECT x
FROM {SelectForm1}<sf:aname>{yname},
     {x}<cm:aname>{yname}
```

Figure 3.32: QSelectP (select painting).

Figure 3.33 shows *QEnableDF*, a query attached to *DeleteForm* in order to enable/disable this form. If the trolley is empty, *DeleteForm* is disabled, as there are no orders to delete.

```
(SELECT count(x)
 FROM {session}<var:trolleyid>{y},
      {y}<ndm:contains>{x}) > 0
```

Figure 3.33: QEnableDF (condition that enables/disables DeleteForm).

Figure 3.34 shows *QSelectDFPn*, a query attached to the control of the form *DeleteFrom* in the main slice of trolley. The query returns the name of the paintings that are in the trolley.

```
SELECT xname
FROM {session}<var:trolleyid>{y},
     {y}<ndm:contains>{x},
     {x}<cm:aname>{xname}
```

Figure 3.34: QSelectDFPn (select paintings (names) that are in the trolley).

Figure 3.35 shows the query *QDeleteO* associated to *DeleteForm* used to delete a selected painting order from trolley. The SeRQL query language was extended with the `DELETE` construct. Basically it is a deletion of statements from an RDF model. The deletion of resources from an RDF model can be easily done by deleting statements of the form

{x}<rdf:type>{rdf:Resource}, where x is the URI of a resource. A garbage collector will make sure that the properties of the deleted resources will be also removed from the model.

```
DELETE
     {x}<ndm:contains>{y},
     {y}<ndm:contained_by>{x},
     {y}<ndm:includes>{z},
     {z}<ndm:included_by>{y},
     {y}<ndm:quantity>{a}
FROM
     {session}<var:trolleyid>{x},
     {DeleteForm1}<df:aname>{yname},
     {y}<cm:aname>{yname},
     {y}<ndm:includes>{z},
     {y}<ndm:quantity>{a}
```

Figure 3.35: QDeleteO (delete selected order from trolley).

In the above queries we did need to extend Se(RQL) with new constructs like URI generators, aggregation functions, and DELETE statements. We do hope that future RDF query languages will be equipped with all these constructs.

3.4.1 Implementation

The implementation of the dynamic variant of the Hera presentation generation phase is based on several data transformations realized in Java. The Se(RQL) queries are executed by Sesame [Aduna, BV, 2005] and the data transformations are implemented in Jena [Hewlett-Packard Development Company, LP, 2005]. In this way the data transformations exploit more of the RDF(S) semantics given by the Hera models than the ones based on XSLT. A transformation language for XML documents like XSLT cannot use the full RDF semantics stored in the RDF/XML serialization of an RDF model.

Figure 3.36 shows the transformation steps for the dynamic variant of the Hera presentation generation. Each transformation step has a label associated with it. Some of these transformations have substeps which are labeled using a second digit notation. In Figure 3.36 there are two types of dashed arrows: “is used by” to express that an RDFS model is used by another RDFS model and “has instance” to denote that an RDFS model has as instance a certain RDF model. A model vocabulary, a model, a model instance, and the generated presentations are depicted by rectangles. The transformation specifications are represented by ovals. In the same way as for the static variant of the implementation models and transformation specifications are classified as application-independent, application-dependent, and query-dependent.

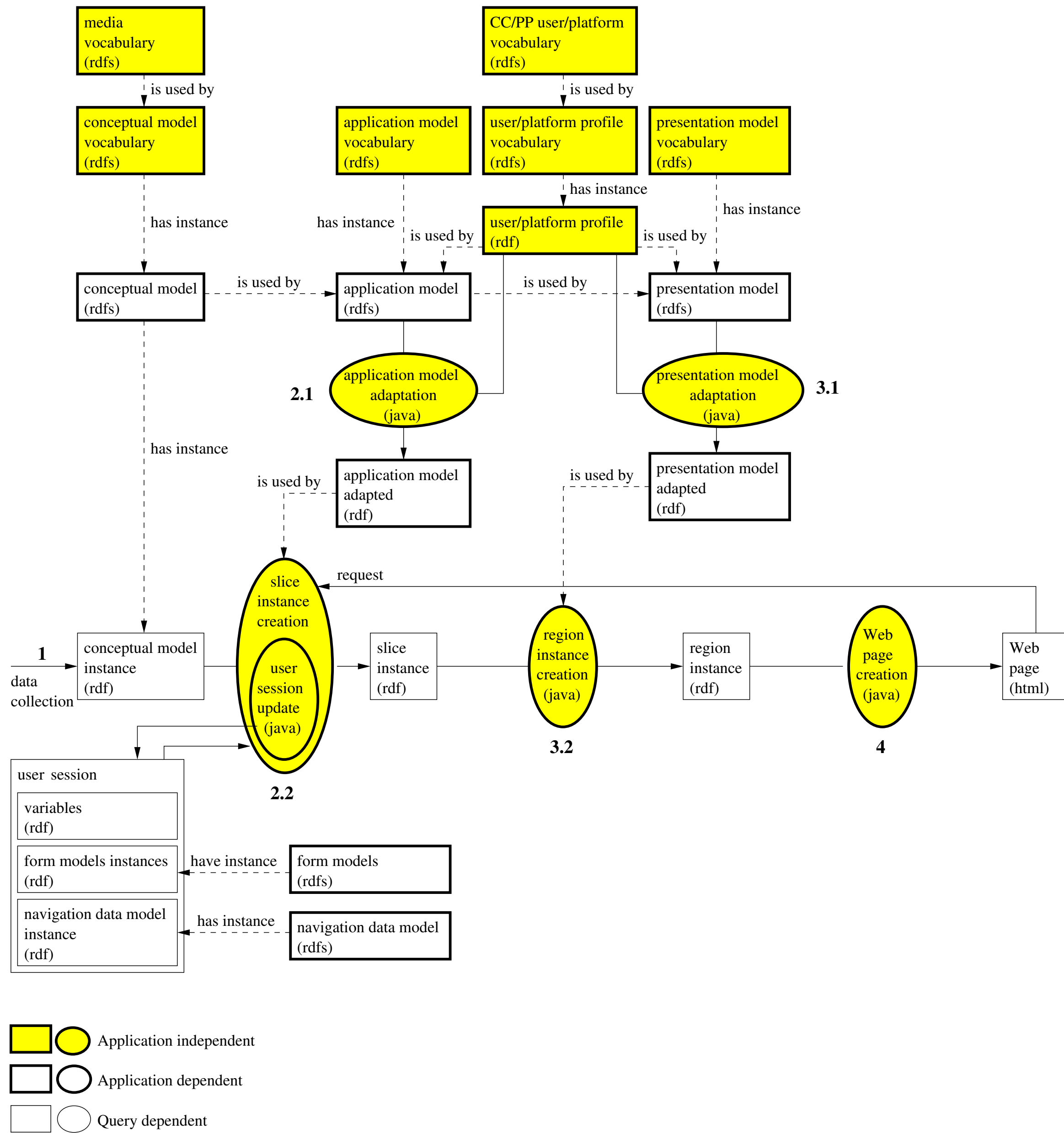

Figure 3.36: Presentation generation using Java.

Step 1, the *data collection* phase, is the same as in the static variant of the implementation. The result of this step is the CMI, i.e., the data retrieved in response to a user query. More information on step 1 can be found in [Vdovjak et al., 2003].

Step 2, the *slice instance generation*, computes a top-level slice instance in response to a user request. This step contains two substeps: the *AM adaptation* and the *slice instance creation*.

Step 2.1, the *AM adaptation*, executes the adaptation specifications on the AM. This transformation has two inputs: the AM and the UP. The UP attributes are replaced in

the conditions by their corresponding values. The slices that have the conditions not valid are discarded and the hyperlinks pointing to these slices are disabled. This step is executed only once at the beginning of a user session. In the current version of the implementation, AM adaptation based on the user model is not performed. Future versions of the implementation, that will make use of the user model will execute this step at each user request.

Step 2.2, the *slice instance creation*, creates the next slice instance. The user request provides: the slice type and the concept instance identifier of the slice instance corresponding to the next Web page to be computed, and possibly form model information, in case that request originates from a form. The first user request in a session specifies also the Hera models that will be used in the current session. The queries associated with the slice navigation that initiated the request and the queries associated to the slice to be computed are executed in the *user session update*. Besides updating the NDMI, the *user session update* also stores in the user session the form models and the value of the variables associated to queries.

Step 3, the *region instance generation*, computes the top-level region instance corresponding to the previously computed slice instance. This step contains two substeps: the *PM adaptation* and the *region instance creation*.

Step 3.1, the *PM adaptation*, executes the adaptation specifications on the PM. This transformation has two inputs: the PM and the UP. The UP attributes are replaced in the conditions by their corresponding values. The layouts and styles that have the conditions not valid are discarded. Similar to step 2.1, this step is executed only once at the beginning of a user session. In the current version of the implementation, PM adaptation based on the UM is not performed. Future versions of the implementation, that will make use of the UM, will execute this step at each user request.

Step 3.2, the *region instance creation*, creates the region instance for the previously computed top-level slice instance.

Step 4, the *Web page creation*, transforms the region instance generated in the previous step into code specific to the user's browser. Note that only one Web page is generated at-a-time. At the current moment only HTML is supported by the implementation.

3.5 Conclusions

Hera is a model-driven methodology for designing Semantic Web Information Systems. The presentation generation phase of the Hera methodology builds a Web presentation for some given input data. The Hera presentation generation phase has two variants: a static one that computes at once a full Web presentation, and a dynamic one that computes one-page-at-a-time by letting the user influence the next Web page to be presented. The design of both variants uses models that are specified in RDF. The implementation of the static variant is based on XSLT data transformations and the implementation of the dynamic variant is based on Java data transformations.

As future work we would like to improve the design and implementation of the Hera

presentation generation phase. For the static variant we would like to implement the CM and media adaptation as given in the design specifications as a separate (from AM adaptation) data transformation. The design of the dynamic variant can be extended by adding specifications for UM-based adaptation. With respect to this we anticipate to reuse some of the work done in the adaptive hypermedia field [De Bra et al., 1999]. The implementation of the dynamic variant needs to be extended with other code generators like HTML+TIME, WML, and SMIL.

Also we would like to investigate the use of a declarative RDF transformation language (similar to XSLT but exploiting better than XSLT the RDF semantics). In [van Ossenbruggen et al., 2005] it is proposed the use of XSLT stylesheets in combination with SeRQL queries (for selections) as a possible RDF transformation language. This hybrid solution is easy to implement and it exploits more of the RDF semantics than XSLT. Nevertheless it relies on the RDF/XML serialization of RDF models and it is less elegant than a solution based on the RDF data model. Lacking an RDF data transformation language based on the RDF data model, we plan investigate the definition and implementation of such a language.

At the current moment Hera doesn't support the requirements phase of the development life cycle of a SWIS. We would like to extend our methodology with a task (activity) model that will specify the activities that can be performed by a user with the system. Once devising a task model one can generate the navigation structure of the application from the task model eliminating the design effort for defining new application models. The task models can be assigned to a particular user or to a group of users (users that share the same task model) facilitating thus the definition of coarse-grained adaptation at navigation level.

Bibliography

Aduna, BV (2005). openrdf.org ... home of sesame. `http://www.openrdf.org/`.

Ayars, J., Bulterman, D., Cohen, A., Day, K., Hodge, E., Hoschka, P., Hyche, E., Jourdan, M., Kim, M., Kubota, K., Lanphier, R., Layaida, N., Michel, T., Newman, D., van Ossenbruggen, J., Rutledge, L., Saccocio, B., Schmitz, P., and ten Kate, W. (2005). Synchronized multimedia integration language (smil 2.0) - [second edition]. W3C Recommendation 07 January 2005. `http://www.w3.org/TR/SMIL/`.

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004). Owl web ontology language reference. W3C Recommendation 10 February 2004. `http://www.w3.org/TR/owl-ref/`.

Beckett, D. (2004). Rdf/xml syntax specification (revised). W3C Recommendation 10 February 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

Bos, B., Celik, T., Hickson, I., and Lie, H. W. (2004). Cascading style sheets, level 2 revision 1 css 2.1 specification. W3C Candidate Recommendation 25 February 2004. `http://www.w3.org/TR/CSS21/`.

Brickley, D. and Guha, R. (2004). Rdf vocabulary description language 1.0: Rdf schema. W3C Recommendation 10 February 2004. `http://www.w3.org/TR/rdf-schema/`.

Brusilovsky, P. (2001). Adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 11(1-2):87–110.

Clark, J. (1999). Xsl transformations (xslt) version 1.0. W3C Recommendation 16 November 1999. `http://www.w3.org/TR/xslt`.

Connolly, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2001). Daml+oil (march 2001) reference description. W3C Note 18 December 2001. `http://www.w3.org/TR/daml+oil-reference`.

De Bra, P., Houben, G. J., and Wu, H. (1999). Aham: A dexter-based reference model for adaptive hypermedia. In *10th ACM conference on Hypertext and Hypermedia (Hypertext'99)*, pages 147–156. ACM.

Diaz, A., Isakowitz, T., Maiorana, V., and Gilabert, G. (1997). Extending the capabilities of rmm: Russian dolls and hypertext. In *30th Hawaii International Conference on System Sciences (HICSS-30)*, volume 6, pages 177–186. IEEE Computer Society.

Dubinko, M., Klotz, L. L., Merrick, R., and Raman, T. V. (2003). Xforms 1.0. W3C Recommendation 14 October 2003. `http://www.w3.org/TR/xforms/`.

Fiala, Z., Frasincar, F., Hinz, M., Houben, G. J., Barna, P., and Meissner, K. (2004). Engineering the presentation layer of adaptable web information systems. In *Web Engineering - 4th International Conference (ICWE 2004)*, volume 3140 of *Lecture Notes in Computer Science*, pages 459–472. Springer.

Fiala, Z., Hinz, M., Meissner, K., and Wehner, F. (2003). A component-based approach for adaptive, dynamic web documents. *Journal of Web Engineering*, 2(1-2):58–73.

Frasincar, F., Barna, P., Houben, G. J., and Fiala, Z. (2004). Adaptation and reuse in designing web information systems. In *International Conference on Information Technology: Coding and Computing (ITCC 2004)*, pages 387–291. IEEE Computer Society.

Frasincar, F. and Houben, G. J. (2001). Xml-based automatic web presentation generation. In *WebNet 2001 World Conference on the WWW and Internet (WebNet 2001)*, pages 372–377. AACE.

Frasincar, F. and Houben, G. J. (2002). Hypermedia presentation adaptation on the semantic web. In *Adaptive Hypermedia and Adaptive Web-Based Systems (AH 2002)*, volume 2347 of *Lecture Notes in Computer Science*, pages 133–142. Springer.

Frasincar, F., Houben, G. J., and Vdovjak, R. (2001). An rmm-based methodology for hypermedia presentation design. In *Advances in Databases and Information Systems (ADBIS 2001)*, volume 2151 of *Lecture Notes in Computer Science*, pages 323–337. Springer.

Hewlett-Packard Development Company, LP (2005). Jena - a semantic web framework for java. `http://jena.sourceforge.net/`.

Houben, G. J., Frasincar, F., Barna, P., and Vdovjak, R. (2004). Engineering the presentation layer of adaptable web information systems. In *Web Engineering - 4th International Conference (ICWE 2004)*, volume 3140 of *Lecture Notes in Computer Science*, pages 60–73. Springer.

Isakowitz, T., Bieber, M., and Vitali, F. (1998). Web information systems. *Communications of the ACM*, 41(1):78–80.

Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., and Scholl, M. (2002). Rql: a declarative query language for rdf. In *Eleventh International World Wide Web Conference (WWW2002)*, pages 592–603. ACM.

Kay, M. (2005a). Saxon (the xslt and xquery processor). `http://saxon.sourceforge.net`.

Kay, M. (2005b). Xsl transformations (xslt) version 2.0. W3C Working Draft 11 February 2005. `http://www.w3.org/TR/xslt20/`.

Klyne, G., Reynolds, F., Woodrow, C., Hidetaka, O., Hjelm, J., Butler, M. H., and Tran, L. (2004). Composite capability/preference profiles (cc/pp): Structure and vocabularies 1.0. W3C Recommendation 15 January 2004.

Lassila, O. and Swick, R. R. (1999). Resource description framework (rdf) model and syntax specification. W3C Recommendation 22 February 1999. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`.

Martinez, J. M. (2003). Mpeg-7 overview. Version 9, ISO/IEC JTC1/SC29/WG11/N5525 March 2003.

Rossi, G., Schwabe, D., and Lyardet, F. (1999). Web application models are more than conceptual models. In *International Workshop on the World-Wide Web and Conceptual Modeling (WWWCM 1999), ER 1999*, volume 1727 of *Lecture Notes in Computer Science*, pages 239–253. Springer.

Rutledge, L., Alberink, M., Brussee, R., Pokraev, S., van Dieten, W., and Veenstra, M. (2003). Finding the story: Broader applicability of semantics and discourse for hypermedia generation. In *ACM Conference on Hypertext and Hypermedia (Hypertext 2003)*, pages 67–76. ACM.

Schmitz, P., Yu, J., and Santangeli, P. (1998). Timed interactive multimedia extensions for html (html+time). W3C Note 18 September 1998. `http://www.w3.org/TR/NOTE-HTMLplusTIME`.

Souchon, N. and Vanderdonckt, J. (2003). A review of xml-compliant user interface description languages. In *International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS 2003)*, volume 2844 of *Lecture Notes in Computer Science*, pages 377–391. Springer.

van Ossenbruggen, J., Hardman, L., and Rutledge, L. (2005). Combining rdf semantics with xml document transformations. *International Journal of Web Engineering and Technology*, 2(4). To appear (guest editors: Frasincar, F., Houben, G. J., and van Ossenbruggen, J.).

Vdovjak, R., Frasincar, F., Houben, G. J., and Barna, P. (2003). Engineering semantic web information systems in hera. *Journal of Web Engineering*, 2(1-2):3–26.

Wireless Application Protocol Forum, Ltd. (2001). Wireless application group: User agent profile. 20 October 2001.