SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–33
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

DOLAR: Virtualizing Heterogeneous Information Spaces to Support their Expansion

Kostas Saidis[1], Yannis Smaragdakis[1,2] and Alex Delis[1]

[1]*Department of Informatics and Telecommunications, University of Athens, 15784, Athens, Greece*
[2]*Department of Computer Science, 140 Governors Drive, University of Massachusetts, Amherst, MA 01003, USA*

SUMMARY

Users expect applications to successfully cope with the expansion of information as necessitated by the continuous inclusion of novel types of content. Given that such content may originate from "not-seen thus far" data collections and/or data sources, the challenging issue is to achieve the return of investment on existing services, adapting to new information without changing existing business-logic implementation. To address this need, we introduce DOLAR, a service-neutral framework which virtualizes the information space to avoid invasive, time-consuming and expensive source-code extensions that frequently break applications. Specifically, DOLAR automates the introduction of new business-logic objects in terms of the proposed virtual "content objects". Such user-specified virtual objects align to storage artifacts and help realize uniform "store-to-user" data-flows atop heterogeneous sources, while offering the reverse "user-to-store" flows with identical effectiveness and ease of use. In addition, the suggested virtual object composition schemes help decouple business-logic from any content origin, storage and/or structural details, allowing applications to support novel types of items without modifying their service provisions. We expect that content-rich applications will benefit from our approach and demonstrate how DOLAR has assisted in the cost-effective development and gradual expansion of a production-quality digital library. Experimentation shows that our approach imposes minimal overheads and DOLAR-based applications scale as well as any underlying datastore(s).

Received . . .

1. INTRODUCTION

The amount of information produced as well as consumed in the world is constantly expanding [1, 2, 3]. In order to cope with such expansion, applications may need to scale-up to support increasing volumes of data. In addition, applications need to gradually expand their information space—the application's private "universe" of data items—to include newly-encountered types of content. As the information space expands, applications have to deal with the following cases:

1. *Support new types of data sources*: Contemporary data-intensive applications such as digital libraries, content management systems and archival repositories, may need to operate atop multiple heterogeneous data sources and, thus, support novel types of datastores. For example, database-oriented applications may need to operate atop XML datastores.
2. *Support new types of data collections*: As applications expand, they may need to support newly-introduced data collections. For example, a community of users may need to use an existing digital library application to introduce and develop a new collection –e.g., by digitizing and documenting real-world artifacts.
3. *Include new items in existing services*: In both of the above cases, the key issue is to efficiently include the new content in any services currently supported. Firstly, applications need to generate new types of business-logic objects to stage the new content items. Secondly, they

have to revise the existing service provision to deal with such new objects and ultimately include the new content in existing services.

Viewed from different perspectives, the above cases raise various multi-disciplinary data integration, data quality and software evolution/adaptation issues [4, 5, 6, 7]. Clearly, developers can handle these cases by code re-engineering. However, expansion requirements can hardly be predicted in detail during the initial application design and development phases. Consequently, the support of a new type of content may break the application, imposing drastic, invasive and expensive source code changes to all service actors. Developers may follow an ad-hoc approach to revisit service actor implementations and they can ultimately succeed in including this new content. However, yet another new requirement for supporting additional types of content may render this approach problematic, breaking service actors yet again. The crucial need here is not to predict the future, but rather to achieve the return of investment on existing services; indeed, the challenge is to enable the existing service provision to operate atop constantly expanding, heterogeneous and diverse information spaces. Thus, a better approach is to base the application on a flexible framework that can isolate the application logic from the type of context and add indirection between the business-logic and the information space. Although adding indirection is a simple idea, designing a general and flexible framework for content expansion is anything but simple. The framework needs to: a) isolate the structure of data, i.e., how the logical organization of data (e.g., the tuples of a database, or the elements of XML documents) map to the application's expectations; b) adapt the physical access to data (e.g., provide network or database connections to objects in a way transparent to the application); c) abstract the object presentation, i.e., smoothly integrate the display of new kinds of objects in the application user interface; d) abstract the object manipulation, i.e., allow new object modification in a uniform way; and e) perform these tasks conveniently and efficiently, in particular without imposing significant runtime overhead over an inflexible, hard-coded implementation of the same features.

In this article, we present *DOLAR (Data Object Language And Runtime)*, a service-neutral virtual information space framework which meets the aforementioned challenges. Employing the separation of concerns principle [8, 9, 10], our approach transcends software, knowledge and data engineering boundaries to virtualize the information space. Specifically, our DOLAR approach provides the following key elements:

- *DOLAR Virtual Objects (DVOs)*: We use DVOs to realize different conceptualizations of data items such as "books", "photos" and "blogs". In contrast to "code objects" composed of properties and methods, DVOs are virtual "content objects" consisting of *Field Sets*, *Relation Contexts*, *Stream Handles* and *Composition Schemes*. DVO specifications are provided in terms of DVO prototypes [11], which are instantiated to offer the business-logic objects at runtime. The unique characteristic of these runtime objects is that they contain no executable code—e.g., they contain no methods—but use composition schemes to model different types/interfaces of data objects. As we show in the article, DVOs are the primary DOLAR mechanism that helps us avoid expensive business-logic code modifications:
  - DVOs enable developers to avoid costly direct coding of business-logic objects. DVO specifications are easy to construct and maintain and allow for the creation of application-specific data-definition utilities. For example, we show how our *DOPsCreator* GUI tool allows us to introduce new business-logic objects without manual coding.
  - DVO composition schemes allow diversely structured data items to expose uniform service-compatible interfaces. As a result, business-logic services can catch up with the addition of new types of data items without any source code modifications.
- *DOLAR Virtual Information Space*: DOLAR virtual information space comprises *DVOStores*, *DVOIndexes* and *DOPSources* in a hierarchical logical space organized in terms of DOLAR *Domains*. These mechanisms help us answer the critical need to not only access but also to modify heterogeneous content in an effective and uniform way. In the context of DOLAR virtual space, developers connect DVOs to datastore artifacts. Although the latter may originate from heterogeneous datastores, the DVOs support uniform "store-to-user" and "user-to-store" information flows, automatically dealing with common tasks including (a) staging the data in

DOI: 10.1002/spe

runtime structures, (b) synchronizing the access to these structures and finally, (c) flushing such structures to underlying datastores as needed. In terms of expressiveness and ease of use, the developer can fetch or store any DVO using literally one line of code, regardless of the data conceptualization, origin and location.

- *Service-neutral DVO API*: DOLAR is realized as a Java class library, fostering the (re)use of DOLAR in different contexts. For example, DOLAR can be used in standalone applications while it may also be part of middleware in distributed applications. To expose the DVO runtime structures to the application-logic services, we use our DVO API. The API offers application-neutrality, as it does not perform any service-specific composition or transformation to the data staged in the DVO structures. Hence, business-logic implementation artifacts, such as modules and components, can synthesize DVOs to cater for any service of choice. Similar application-independence is found in database systems, where the result sets returned by SQL queries are made available to applications in terms of runtime structures that stage involved database tuples. A database system makes no assumptions about the actual usage of data by the business-logic, offering a general-purpose system for managing relational data in terms of any application. Respectively, DOLAR offers a service-neutral framework which virtualizes information spaces to facilitate their efficient expansion.

To keep expansion costs in check and avoid expensive source-code revisions, the above DOLAR mechanisms enable applications to extend their "low-level" information space options without modifying their "high-level" business-logic services. As we show in the article, our DOLAR approach has enabled the cost-effective construction and gradual expansion of the Pergamos information space. Pergamos has been in production for nearly five years and is currently the largest academic digital library in Greece hosting about 300,000 items and exceeding $1\,TB$ of space. In particular, we present how the use of DOLAR has helped us cope with the dual pressure of gradually (a) using Pergamos to develop a variety of collections originating from independent digitization projects at the University, (b) adding existing University collections in Pergamos, including "books", Domino-based "theses", technical reports etc. Moreover, in our experimental evaluation, we show that DOLAR-imposed operational overheads are minimal and DOLAR-based applications scale as well as the underlying datastore(s).

The remainder of this article is organized as follows. Section 2 motivates the design of DOLAR's virtual information space and Section 3 presents its elements. Section 4 discusses the role of our proposed virtual objects and their operation. Sections 5 and 6 illustrate the DOLAR-based implementation of Pergamos content presentation, curation and search services. The evaluation of our approach in terms of effectiveness and efficiency is discussed in Section 7. Section 8 discusses related work and finally, Section 9 offers our conclusions and future work.

2. MOTIVATING EXAMPLE

We use the Pergamos expansion requirements to motivate our discussion for virtualizing the information space. Given that Pergamos can be classified as a centralized, multi-tier, web application, we elaborate on modern service architectures, using Model-View-Controller (MVC) [12] as a vehicle for our discussion. MVC is a well-established pattern that effectively separates data from presentation and is routinely used in multi-tier, enterprise-scale applications [13, 14]. According to MVC, services act as "controllers" of the information flow issued between "view" and "model" components. The role of "model" components is to offer the business-logic objects required for staging the data at runtime. Such "model" components may employ "data access" actors to interact with underlying storage facilities. In turn, the role of "view" components is to offer user-oriented views of information. For example, such "view" actors may employ HTML displays or GUI components like list-boxes to present the data to the user. Finally, "controller" components realize service provision entry points, composing "model" and "view" actors. Regardless of the terminology in place, architectures use a similar set of actors to realize service provision and, clearly, any other architecture can be considered in a similar manner.

Prepared using speauth.cls
DOI: 10.1002/spe

2.1. Pergamos Business Case

Pergamos plays a dual role as it publishes Univ. of Athens digital collections for web visitors, and offers a platform for members of the community to develop and document digital collections [15, 16]. This duality of business requirements is addressed by Pergamos front and back-end subsystems. Pergamos back-end subsystem offers an authoritative and effective content documentation infrastructure. Experts in each collection's domain are involved in defining documentation details and specifying the particular descriptive metadata to be used for each new collection. This yields a collection hierarchy comprising diverse content, such as the *Historical Archive*, the *Folklore* as well as the *Theatrical* collections to name a few. In addition, Pergamos user-interfaces and their ease-of-use are of key importance as domain experts along with researchers, students and library staff routinely use such interfaces to document content items. Supporting the hierarchical nature of our collections, Pergamos offers collection navigation services. During collection browsing, our front-end services supply users with HTML displays of the collection hierarchy. The browsing of our back-end subsystem provides additional web-based forms to our authorized users. These forms allow for item editing, addition of new items as well as deletion of obsolete ones. Finally, our services support an information space search capability, collectively spreading over both front and back-end subsystems.

The need for information space extension was the dominant Pergamos requirement. On one hand, new digitization projects emerged, including the *Museum of Mineralogy* and the *Byzantine Music Manuscripts* collections. The critical issue here was to supply groups of domain experts and digitization workers with effective data ingestion and curation services at the get-go of their work, even though each digitization group worked on different collection development projects. On the other hand, the need to include existing heterogeneous collections of the University occurred, such as the *Anthemion* "books" database and the Domino-based "theses". Here the key issue was to include such collections in our front- and back-end subsystems without "breaking" existing code. In both cases, the challenge was to reduce the time and effort required to make our Pergamos services adapt to the needs of each new collection.

2.2. Virtualizing the Information Space

To make the Pergamos subsystems catch up with new collection requirements in a timely fashion, we needed to avoid expensive code re-engineering. To this end, it was crucial to separate the "low-level" information space idiosyncrasies from the "high-level" service provision logic. To achieve this separation, we had to deal with the following *four information management options* [17]:

1. *information discovery options* correspond to the indexing and/or searching dimension of an application (i.e., how the data is being indexed/searched) ,
2. *information access options* reflect the information accessing and storing dimension of an application (i.e., how the data is being accessed/stored),
3. *information conceptualization options* correspond to conceptualizations used by the business-logic of an application (i.e., how the data is being staged at runtime), and
4. *information utilization options* reflect the synthesis of information in the context of an application (i.e., how the data is being composed to offer end-user services).

Figure 1 depicts the composition of the information space made-up of the above four options. Clearly, business-logic should be separated from any information access/storage options; otherwise service-provision would become coupled to the particular datastore used beneath. To this end, we use our proposed virtual objects to separate the handling of information space options (2) and (3), offering a unified mechanism for staging heterogeneous data at runtime. For example, in Pergamos, although the data storage formats vary among collections, all collection items are staged in terms of uniform DVO-based runtime structures. In addition, to help the business-logic operate in isolation of the structural diversity of data, we need to supply services with a uniform interface to the staged data. To this end, we use our proposed composition schemes to separate the handling of information space options (3) and (4) and hide any data-inherent differences from service provision components. For example, in Pergamos we use composition schemes to offer data-entry facilities that adapt to each collection requirements. Finally, schemes also allow us to index diverse data in a uniform

Prepared using speauth.cls
DOI: 10.1002/spe

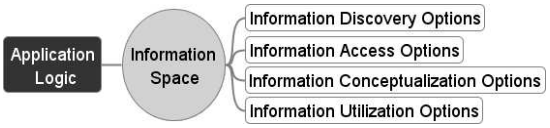


Figure 1. Information spaces consist of multiple information management dimensions

manner, helping us to separate the handling of information space options (1) and (3). For example, we show how Pergamos business-logic uses composition schemes to decide whether and how to index each different collection item.

In order to facilitate the expansion of the information space, the business-logic should be separated from the information space in terms of all four dimensions of Figure 1. Otherwise, the expansion will cause costly and invasive changes to the business-logic code. To illustrate this, consider the MVC service architecture operating atop our Pergamos "photo-album" items. Here, a "data access" (or similar-in-nature) actor will be used to wrap the underlying XML repository. At the same time, the "model" actors will offer "album" and "photo" business-logic objects. In turn, these objects will be synthesized by "view" actors to yield user-consumable views of "photo-album" items. However, in order to realize service provision, the above (or similar) service actors tend to become intertwined with the specific content. This coupling is due to the "tyranny of dominant decomposition"; by and large, any architectural choice or pattern separates concerns in terms of a single dimension at a time [10]. For instance, although MVC enables applications to separate data from presentation, it does so in the context of a given set of data at a time. To foster information expansion, we need to exploit the benefits of MVC (or any other service architecture adopted), yet without realizing a different instance of the architecture to handle each different type of content. For example, imagine that the above Pergamos MVC architecture is augmented to also support the Anthemion collection of digitized "books". The latter originate from a new source –a network SQL database– requiring the application to extend its information access options. "Book" artifacts will likely be indexed and searched differently, requiring an extension of the application's information discovery options. New kind of business-logic conceptualizations will also be necessary to stage the "book" items at runtime. Finally, to incorporate "books" in the service provision, the application has to extend its information utilization options, revisiting its "controller", "view" and similar components. Clearly, having support for new types of items is scattered throughout all service actors, causing individual actor implementations to become entangled. Scattering occurs when a single requirement affects multiple components and entanglement appears when multiple requirements are interleaved within a single component, leading to crosscutting of concerns [10, 18].

This crosscutting of concerns is an apparent obstacle as each additional need to expand the information space imposes substantial and costly changes in all business-logic actors. To curtail these expenses, our approach *virtualizes* the information space.

3. DOLAR'S VIRTUAL INFORMATION SPACE

The DOLAR virtual information space is organized in terms of *domains*. Our DOLAR *Dictionary* serves as the "domain of domains" and helps combine diverse information contexts in a single logical hierarchical space.* Within each DOLAR domain, we support unique realizations of the four information management options discussed in the previous section. Our virtual information space allows business-logic to dissociate the four information management options of Figure 1. More specifically:

*the term "domain of domains" does not refer to the upper ontology of ontology-oriented approaches. DOLAR domains are simply used to identify distinct namespaces, defining the scope of proposed DOIndexes, DOStores or DOPSources. To this effect, the term "domain of domains" is used to reflect the hierarchical nature of DOLAR's virtual space.

Prepared using speauth.cls
DOI: 10.1002/spe

1. the information discovery options are realized through the *DOIndex* mechanism which wraps application-specific indexing/searching facilities. The *DOIndex* provides a way to automatically index new types of content and also display such new content in search results.
2. information access options are managed in terms of the *DOStore* mechanism. This mechanism is placed between "model" components and underlying datastores and exposes a uniform interface, regardless of the conceptualizations and the storage/access details involved. This enables applications to extend their information-access and information-conceptualization options independently of each other.
3. information conceptualizations are expressed in terms of *DVO Prototypes (DOPs)* and are loaded from *DOPSources*. DVOs and DOPs automate the generation of business-logic objects.
4. information utilization options are managed in terms of *DVO composition schemes*, offering an effective means to compose data-inherent and application-inherent behavior. Schemes are unique elements of our proposal and we discuss them at length in Section 4.

A DOLAR domain may consist of any combination of specific DOStore, DOIndex and DOPSource elements, while a DOLAR space (also called DOLAR dictionary) may consist of one or more domains. The setup of the DOLAR space and its constituent domains depends on the particular application needs. For example, our Pergamos back-end and front-end services operate atop three datastores. These are the Pergamos-internal XML repository, the relational solution that holds the *Anthemion* collection of books and, finally, the Domino document database. We also have numerous DOP definitions for expressing business-logic conceptualizations for content items such as "books", "theses", "photo albums" and "folklore-artifacts". Moreover, we employ two different information indexing/searching facilities: we use a Lucene full-text index for enabling user free-text search and a relational database for offering field-oriented search.

Figure 2a shows a logical view of the virtual information space we have created in Pergamos; the DOLAR dictionary includes here three domains, namely **`lib.uoa.gr`**, **`history.uoa.gr`** and **`law.uoa.gr`**. Due to the dominance of Internet-like identifiers, we use the **`lib.uoa.gr`** form for naming DOLAR domains throughout this paper. It is clear, though, that DOLAR domains do not stand for physical network hosts but represent logical information contexts, defining the scope of DOStore, DOIndex and DOPSource elements in the context of DOLAR's unified namespace. As Figure 2a depicts, the **`history.uoa.gr`** domain provides **`anthemion`** DOStore driver which wraps the database holding the *Anthemion* collection. The **`law.uoa.gr`** DOLAR domain provides the **`domino`** DOStore driver of the Domino-based theses. The dictionary also contains a **`lib.uoa.gr`** domain, providing: (a) the **`pergamos`** *DOStore*, wrapping the Pergamos XML repository, (b) the **`main`** *DOPSource* issued as the central source of DOP definitions and (c) the two *DOIndex* elements, namely **`fullText`** and **`dc`**, wrapping the aforementioned full-text index and database respectively. That is, the first two domains contain the mechanisms needed to access two separate datastores, while the third domain contains not just the mechanism to access a third datastore but also information describing the data schema for all three datastores. In another context, involved *DOStore*, *DOIndex* and *DOPSource* elements could have been meshed in domains differently.

Figure 2b depicts an architectural view of the virtual information space in Pergamos, showing the separation of the four information management options in terms of our low-level **`DOStore`**, **`DOIndex`** and **`DOPSource`** APIs and our high-level usage **`Dictionary/Runtime`** and **`DVO`** APIs. Our Dictionary/Runtime API of Figure 2c yields an "operational" view of DOLAR's virtual space elements, where a DOLAR domain acts as a registry of *DOStore*, *DOIndex* and *DOPSource* elements, and the dictionary acts as the registry of domains. Using the API, developers can query the DOLAR dictionary to obtain the list of currently registered elements, while they can also extend the dictionary dynamically, by adding new domains or augmenting existing domains with new *DOStore*, *DOIndex* and *DOPSource* elements.

The virtualization of the information space is achieved by realizing all DOLAR domain, *DOStore*, *DOIndex*, *DOPSource*, DOP elements as well as DVOs as first-class objects of the DOLAR dictionary. This yields a common memory space which transcends physical/network boundaries, hiding any data origin details and offering a unified virtual information space. Individual elements of this virtual space are being addressed through DOLAR URIs of the following form:

Prepared using speauth.cls
DOI: 10.1002/spe

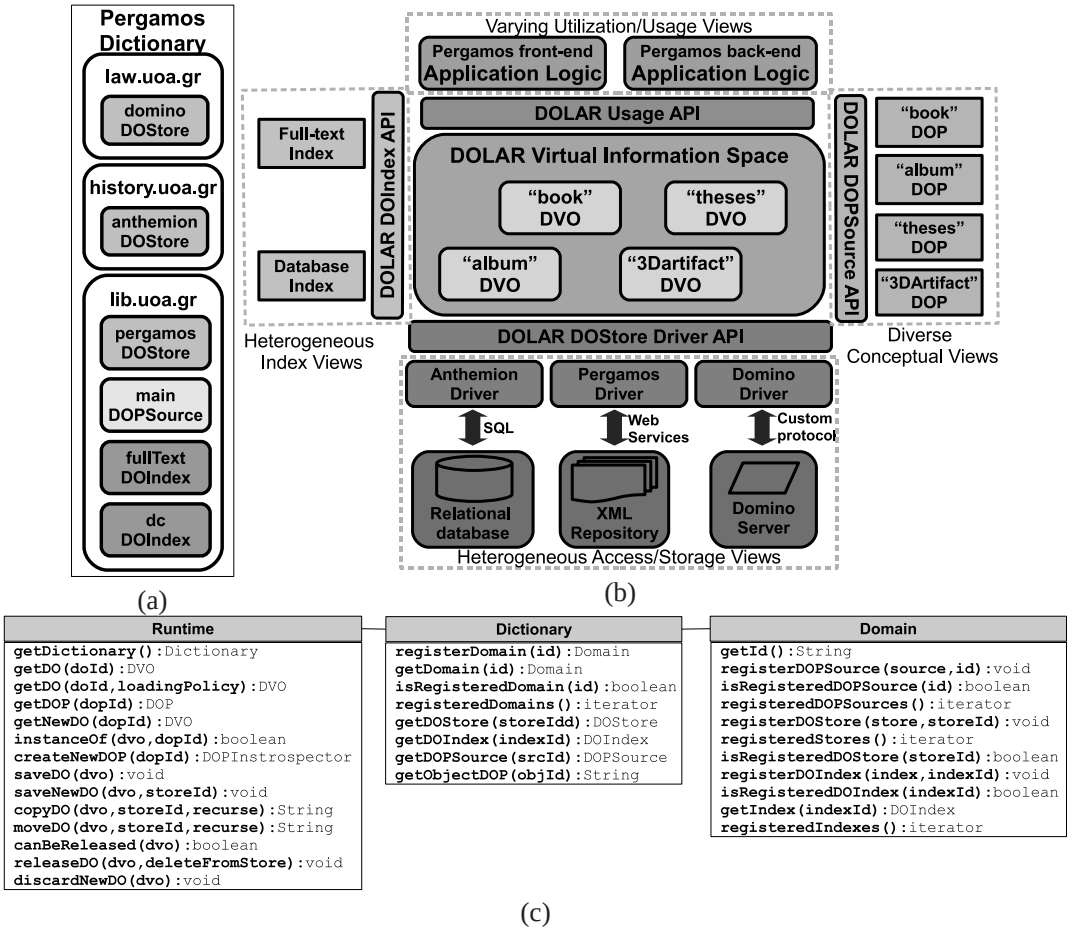


Figure 2. DOLAR Virtual Information Space: (a) A logical view of Pergamos Virtual Information Space (b) An architectural view of Pergamos Virtual Information Space (c) DOLAR Dictionary/Runtime API: the operational view of the Virtual Information Space

```
dolar://domainId/storeId/itemId
dolar://domainId/dop/dopSourceId/dopId
dolar://domainId/index/indexId
```

Resolving such identifiers involves DOLAR performing an automated virtual space lookup, resembling a "pointer dereference" procedure issued by a programming language at runtime. For example, `dolar://lib.uoa.gr` will either resolve to a domain termed `lib.uoa.gr` or a `NotFound` error will be thrown. In turn, `dolar://lib.uoa.gr/pergamos` resolves to our Pergamos XML-based DOStore driver, while `dolar://history.uoa.gr/anthemion` identifies the DOStore driver of the *Anthemion* database collection. We also use DOLAR URIs to identify content items. For example, `dolar://lib.uoa.gr/pergamos/album:100` refers to a "photo-album" XML item, while `dolar://history.uoa.gr/anthemion/book:12` refers to a "book" database item. Automation is achieved through our virtual object metaphor. When developers resolve such content item identifiers, DOLAR provides "ready-made" virtual objects that stage the data held in the "album:100" XML item and "book:12" database item respectively.

4. DOLAR VIRTUAL OBJECTS

We use virtual objects to automate the process of expanding the information space with new business-logic conceptualizations. We view storage artifacts as serializations of virtual objects and

Prepared using speauth.cls
DOI: 10.1002/spe

so we separate the information that makes up a conceptualization from any of its serializations. A virtual object (DVO) is defined as follows:

$$DVO = \{DOP, DOStoreDriver, storedItem\}$$

where *DOP* defines the logical structure of the virtual object, *storedItem* designates a storage artifact (i.e., a set of database records, an XML document, etc.) and the *DOStore* driver is the bidirectional mechanism that helps retrieve/store the data in question. In this section, we discuss all pertinent mechanisms that collectively produce virtual objects (DVOs) at runtime and outline their salient operations.

4.1. Virtual Object Data Model

Virtual objects offer a logical view of the data held in storage artifacts and are composed of *Field Set, Relation Context, Stream* and *Scheme* elements. Figure 3a reflects this layout.

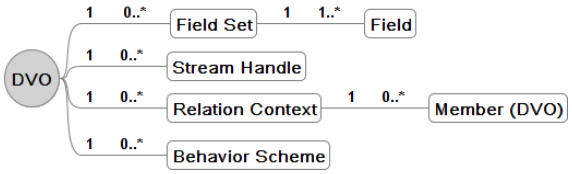


Figure 3. The conceptual/logical view of a virtual object

The structure of individual virtual objects—the names as well as the types of elements *DVO*s contain— is governed by the user's specification defined in isolation from any specific storage details. In particular, the DVO data model consists of:

- *Field Sets:* they refer to field-like data such as name/value pairs, database tuples, XML-encoded metadata or any other form of named attributes. A virtual object specification may contain zero or more *FieldSet* definitions, each one consisting of one or more *Fields*. We make no assumptions about the storage of these fields or their conformance to any standard. For example, the Pergamos-internal repository stores Dublin Core (DC) [19] metadata in XML-encoded form, while the Anthemion collection holds its custom "book" fields in a relational database. Thus, a *FieldSet* definition designates a set of fields held in a storage artifact, regardless of any storage details.
- *Relationships:* Relationships among content items are expressed as relation-contexts. Virtual object specifications may contain multiple *Relation Context* definitions, each one used to outline a particular relationship among items. During specification, the *RelationContext* definition provides the types of objects that can participate in a given relationship, yet, without making any assumptions about the storage representation of such a relationship. For example, Anthemion stores the book-to-page relationships in a database table, while in Pergamos XML repository, object-to-object relationships are held in terms of RDF triples.
- *Stream Handles:* We use our stream-handles to model any underlying locally or remotely-stored "document-based" digital content. For example, the PDF document holding the full-text of a thesis is modeled by a stream handle in DOLAR. At specification time, a *StreamHandle* definition provides the MIME types supported by the underlying "document".
- *Composition Schemes:* we use our composition schemes to offer runtime projections of virtual objects. The above *Field Sets, Relation Contexts* and *Stream Handles* designate the internal state of a virtual object. Composition schemes help us expose this internal state to the service actors.

Figure 4 presents five examples of virtual objects. The "album" object of Figure 4a stands for a "photo album" XML item originating from ceremonies of the Univ. of Athens. Such items comprise descriptive metadata and the album's digitized photographs. Metadata is represented in terms of a **`dc`** *FieldSet* and its respective *Fields*, while a *RelationContext* termed **`structure`** is used to represent the containment relationship between "album" and "photo" objects. "Photo" objects hold three versions of the digitized photograph in terms of *Stream Handles*: a high resolution image (hq), a web-quality image (web) and a thumbnail image (thumb). In similar fashion, Figure 4b

Prepared using speauth.cls
DOI: 10.1002/spe

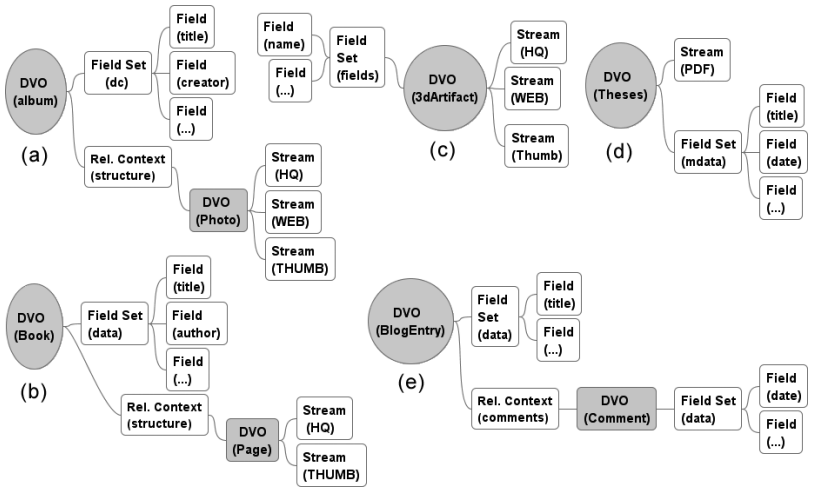


Figure 4. Various content item conceptualizations expressed in terms of DOLAR's virtual objects

depicts a "book" item consisting of "page" items. These items originate from the database holding the *Anthemion* collection which abides to the following SQL schema: `book(id,title,author,...)`, `page(id,bookId,order,tiffUrl,tiffLength,jpegUrl,jpegLength)`. Figure 4c shows a "folklore-artifact" object originating from the Folklore Collection, used for representing digitized 3D folklore artifacts. The "theses" object of Figure 4d refers to the Domino-based theses items, comprising the full text of the theses and its descriptive metadata. DVOs can express diverse conceptualizations including the blog entries and their respective comments modeled by the "blog-entry" object of Figure 4e.

Composition schemes help us hide any data-inherent idiosyncrasies from service actors. For instance, even though both "photo-album" and "folklore-artifact" items of Figure 4 contain thumbnails, the thumbnail image for a "photo album" is derived from the album's first digitized photograph, while the thumbnail for a "folklore-artifact" is an image of the digitized artifact itself. To foster information expansion, we need to disassociate service actors from such data-inherent details. As far as service implementation is concerned, the information about how the thumbnail originated from a "photo-album", a "folklore-artifact" or any other type of items should be transparent. The interpretation of the behavior of an item (i.e., how to acquire the thumbnail) depends on the structure of the content item at hand. The composition of the item's behavior (i.e., what to do with the thumbnail) depends on the details of the service provision at hand. Thus, we introduce the following distinction:

- *Data-inherent behavior* depends on the structure and specific characteristics of the item at hand. An example of data-inherent behavior was the above different treatment of "photo-album" and "folklore-artifact" thumbnails.
- *Application-inherent behavior* depends on the overall functionality supported by an application, including communication mechanisms, user interfaces and other service provision features, regardless of the type of content items. For example, an application will either support HTML display for all its types of items or it will not support such a display at all.

Composition schemes designate a bridge between (a) the application-inherent behavior, which is realized by service actors and (b) the data-inherent behavior, which is realized by DVO runtime views. This helps us offer a uniform, scheme-based DVO interface to service actors, which hides any data-inherent details and ultimately enables us to expand the information space without modifying the implementation of service actors.

4.2. Defining Virtual Object Specifications

A significant cost involved in expanding an application originates from the generation of new types of business-logic objects –the "model" actors in the MVC terminology. Virtual objects can

Prepared using speauth.cls
DOI: 10.1002/spe

significantly reduce such "model" generation costs and help developers avoid error-prone manual coding. In particular, DOLAR offers a DVO Introspection API for specifying the structure and layout of virtual objects, providing an effective and lightweight data-definition mechanism which operates in isolation of any data-storage details. For example, using this API, the specification of the "photo-album" conceptualization of Figure 4a is defined as follows :

```
DOP photo = DOLAR.newPrototype("photo");
photo.setLabel("en", "Photograph");
photo.addStreamHandle("HQ", ["image/tiff"]);
...
DOP album = DOLAR.newPrototype("album");
album.setLabel("en", "Ceremony Photo Album");
album.addFieldSet("dc");
album.setLabel("dc", "en", "Descriptive Metadata");
album.addField("dc", "title", String, MULTILINGUAL+MANDATORY);
album.setLabel("dc.title", "en", "Title");
album.addField("dc", "description", String, MULTILINGUAL+MANDATORY);
...
album.addRelationContext("structure", ["photo"]);
album.seal();
```

As shown above, *Field Set*, *Field*, *Stream*, *Relation Context* and *Composition Scheme* specifications can be supplied with multi-lingual labels and descriptions, assisting applications to render human-consumable representations of virtual objects in an effective manner. For instance in Pergamos, we use such labels and descriptions to present individual DVO elements in terms of web-form fields. DOLAR also supports multilingual values for its *Field* elements. For example, the "title" field defined above can hold different values for different languages.

In brief, the DVO Introspection API provides a set of methods for defining *Field, Stream, Relation Context* and *Composition Scheme* elements, supporting the "sealing" of DVO prototypes to prevent any further structural modifications. A significant benefit of the DVO Introspection API is that it allows for the creation of application-specific data-definition tools. In Pergamos we have used this API to create *DOPs Creator*, a GUI-tool for the definition of virtual object specifications without manual coding. Figure 5a shows the creation of the "photo-album" conceptualization with the help of this tool. DOPs Creator encodes virtual object definitions as XML files; Figure 5b depicts the XML definition of these "album" and "photo" virtual object specifications. Such XML representations provide the default DOLAR serialization format of virtual object specifications.

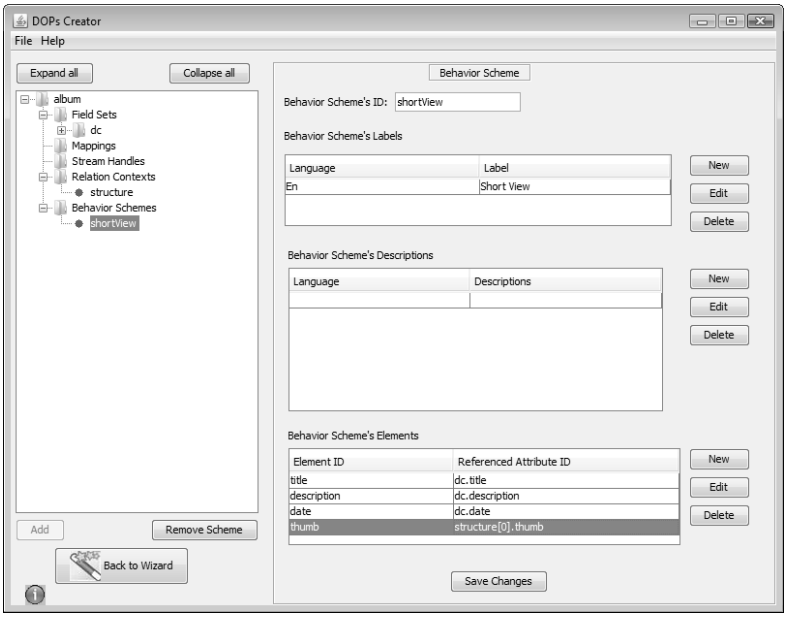
4.3. Loading DVO Prototypes in the DOLAR Space

When it comes to the operational aspects of virtual objects, DOP definitions may originate from various sources. To this end, we use the *DOPSource* mechanism to support any facility that can hold DOP definitions. For example, an application that uses our default XML-based DOP definitions may use its host file-system as a DOP source. In Pergamos, we use this approach as DOP definitions are locally held in the file-system of our front-end and back-end hosts. Our **`FileSystemDOPSource`**—the **`main`** DOPSource in Pergamos DOLAR dictionary of Figure 2a—loads XML-based DOP definitions using the file name to identify the enclosed virtual object specification. More specifically, the "album" DOP definition of Figure 5 is placed in a file named **`dop.album.xml`**; in the virtual information space the "album" DOP definition is then referenced through the **`dolar://lib.uoa.gr/dop/main/album`** URI. Our **`DOPSource`** API features three operations:

1. **`containsDOP(dopId)`**: indicates whether the DOPSource contains a DOP definition identified by the given dopId,
2. **`listDOPs()`**: offers a list of the currently held DOPs,
3. **`loadDOP(dopId)`**: loads the DOP definition identified by the given dopId.

Applications can realize these operations to load their virtual object specifications from any physical/network locations; an application may utilize DOLAR's **`DOPSource`** and **`DVOIntrospection`** APIs to load XML DOP definitions over the web, for example. In general, to support information expansion, applications can augment the virtual information space with a new DOP source, or augment existing DOP sources with new DOP definitions. In the next section, we discuss the processing of DOP definitions that DOLAR carries out in order to instantiate *DVO*s.

DOI: 10.1002/spe



(a) The DOPs Creator GUI for issuing virtual object specifications

```
<dop id="album">
 <label lang="en">Ceremony Photo Album</label>
 <fields>
  <set id="dc">
  <label lang="en">Descriptive Metadata</label
  <field id="title" isMandatory="true">
   <label lang="en">Title</label>
  </field>
  <field id="description" isMandatory="true">
   <label lang="en">Description</label>
  </field>
  ...
  </set>
 </fields>
 <relations>
  <context id="structure">
  <label lang="en">Photographs</label>
   <target dop="photo"/>
  </context>
 </relations>
 <composition>
   <scheme id="shortView">
   <label lang="en">Short View</label>
   <element id="title" ref="dc.title"/>
   <element id="description" ref="dc.description"/>
   <element id="date" ref="dc.date"/>
   <element id="thumb" ref="structure[0].thumb"/>
  </scheme>
  ...
 </composition>
</dop>
```

```
<dop id="photo">
 <label lang="en">Photograph</label>
 <digitalContent>
  <streamHandle id="hq">
   <label lang="en">
    High quality image
   </label>
   <mime type="image/tiff"/>
  </streamHandle>
  <streamHandle id="web">
   <label lang="en">
    Web quality image
   </label>
   <mime type="image/jpeg"/>
  </streamHandle>
  <streamHandle id="thumb">
   <label lang="en">
    Thumbnail image
   </label>
   <mime type="image/jpeg"/>
  </streamHandle>
 </digitalContent>
</dop>
```

(b) XML definitions of “photo” and “album” virtual objects generated by the tool

Figure 5. Defining Virtual Object Specifications with the DOPs Creator Tool

4.4. Virtual Object Instantiation: Automated “Model” Actors

Virtual objects automatically realize user conceptualizations at runtime as DOPs and DVOs share a class-to-object relationship. During instantiation, DOLAR processes DOP definitions and produces runtime virtual objects with corresponding layout in terms of **`FieldSet`**, **`Field`**, **`Stream`**, **`RelationContext`** and **`Scheme`** DVO API runtime structures of Figure 6.

Once virtual object specifications are in place, developers acquire virtual objects using the **`runtime.getDO`** Dictionary/Runtime API call, supplying the DOLAR URI of the content item of interest. For example, the calls:

```
DVO album = runtime.getDO("dolar://lib.uoa.gr/pergamos/album:100")
DVO book = runtime.getDO("dolar://history.uoa.gr/anthemion/book:12")
```

provide **`album`** and **`book`** virtual objects automatically, each one corresponding to the underlying “photo-album” XML item and “book” database item respectively.

DOI: 10.1002/spe

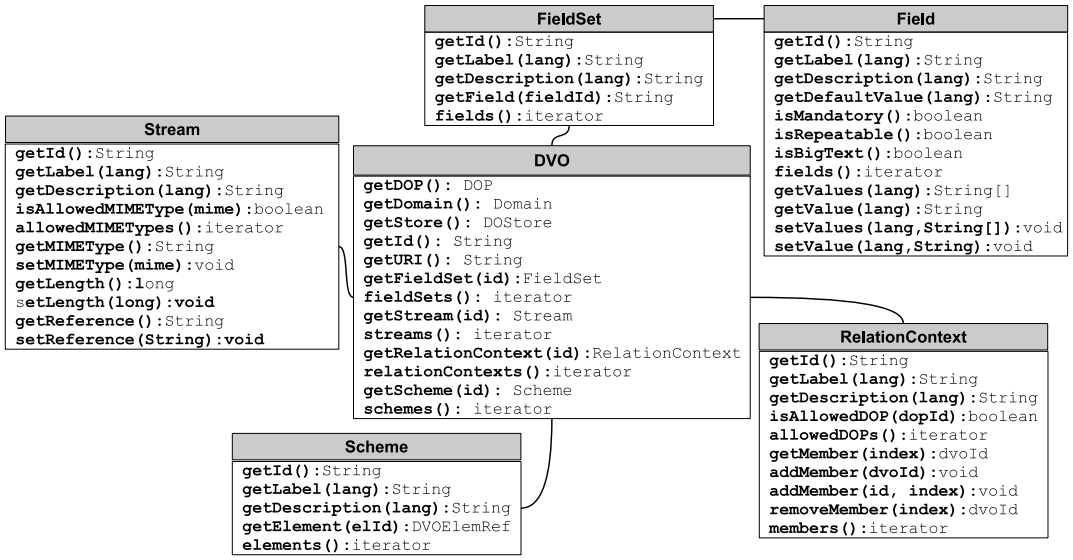


Figure 6. The runtime view of a virtual object in terms of the DVO API

During instantiation, newly created virtual objects are connected to storage items. This is performed through the *DOStore* driver that participates in the DOLAR URI; the **`album:100`** DVO will be connected to the underlying "photo-album" using the **`pergamos`** driver, while the **`book:12`** DVO is linked to the underlying "book" item through the **`anthemion`** driver. Instantiation "bridges" the logical context of virtual object specifications with the storage-specific context of a *DOStore* driver to offer a runtime context provided by the newly instantiated DVO. From a developer's perspective, acquiring a virtual object is simply equivalent to resolving a DOLAR identifier. In a uniform and automated fashion, applications can obtain any virtual objects, originating from any heterogeneous datastores.

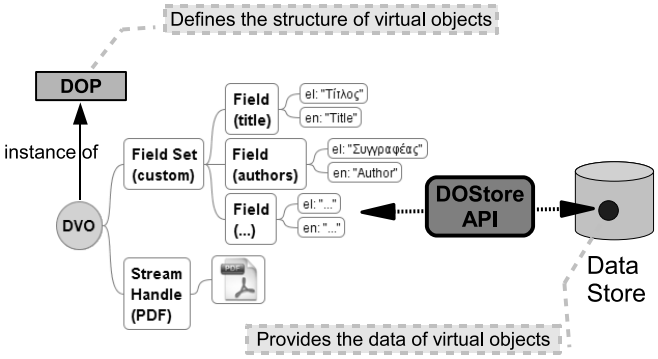
4.5. Two-way Linking of Virtual Objects to Any Storage Artifacts

Developers use the *DVO*-API to fetch any data originating from the stored content space. They also use the *DVO* API to modify DVO data and ultimately use the **`runtime.saveDO`** Dictionary/Runtime API call to save DVOs back to persistent storage. The role of our *DOStore* mechanism is to supply virtual objects with storage-independence.

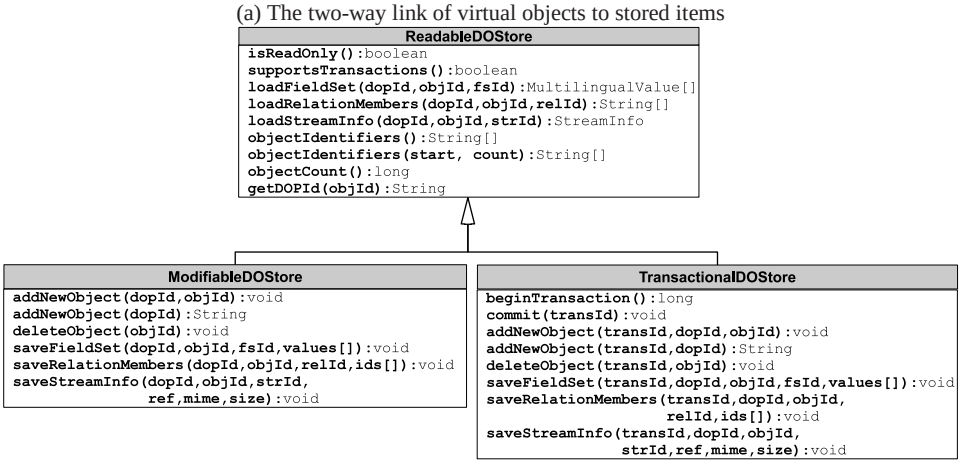
Figure 7a depicts the two-way link between a virtual object and a stored item. To realize this link, the DVO API employs our *DOStore* mechanism to function "behind the scenes". This mechanism essentially realizes a bidirectional connection between the **`FieldSet`**, **`Field`**, **`Stream`** and **`RelationContext`** runtime DVO structures and any storage structures found in the underlying datastores. Figure 7b depicts our three *DOStore* API interfaces. These interfaces offer a unified virtual object store API which allows DVOs to operate atop heterogeneous stores in a uniform manner. Firstly, the **`ReadableDOStore`** interface defines the essential data fetching operations performed by virtual objects to load data in their **`FieldSet`**, **`Stream`** and **`RelationContext`** structures. The **`ModifiableDOStore`** interface extends **`ReadableDOStore`** to provide the essential data insert, update and delete operations performed by virtual objects, while **`TransactionalDOStore`** extends **`ReadableDOStore`** to allow our virtual object environment to perform data modification in a transactional fashion, if transactions are supported by the underlying datastore. Thus, a DOStore driver implementation can be realized in three ways, each one reflecting the specific data-store choice beneath:

- `StoreDriverA` **implements** *ReadableDOStore*: this type of driver refers to read-only data sources, such as a web-based source or any other read-only data source available in an operational environment.

DOI: 10.1002/spe



(a) The two-way link of virtual objects to stored items



(b) The DOStore API

Figure 7. The DOStore Mechanism

- `StoreDriverB` **implements** *ReadableDOStore*, *ModifiableDOStore*: such a driver encapsulates a modifiable data source that does not support transactions, such as a file-based XML store, for example.
- `StoreDriverC` **implements** *ReadableDOStore*, *TransactionalDOStore*: this driver wraps a transactional data source, such as a relational database.

DVO API Implementation: Here, we discuss the DVO API implementation, showing how DVOs use the above three interfaces for staging, modifying and inserting data in a fashion transparent to the developer.

◇ Staging Data: Virtual objects use the **`ReadableDOStore`** API to interface with their corresponding drivers and stage underlying data. Specifically, the first time a developer issues a **`DVO.getFieldSet()`** call to a virtual object, the object will call the **`ReadableDOStore.loadFieldSet()`** method "behind the scenes" to contact the underlying datastore and stage field-like data in the form of **`Field`** structures. Respectively, when the developer issues a **`DVO.getRelationContext()`** call for the first time, the given DVO will use its *DOStore* driver to load the members of the relationship by invoking the **`ReadableDOStore.loadRelationMembers()`** method. Finally, the first time a developer issues a **`getStream()`** DVO API call, the DVO will transparently call the **`ReadableDOStore.loadStreamInfo()`** method of its *DOStore* driver to load the underlying stream/file information. In all cases, any subsequent calls for an already-loaded field-set, relationship or stream handle will not result in contacting the underlying data source, as the DVO keeps track of the loaded elements internally.

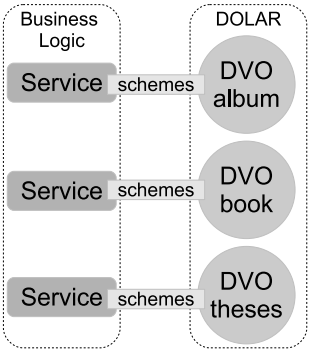
◇ Modifying Data: Developers *modify* DVO-entailed data by using the DVO API. In particular, the **`Field`**'s **`setValue()`** or **`setValues()`** calls replace the runtime values held in **`Field`** structures. The **`Stream`**'s **`setReference()`**, **`setMIME()`** and **`setLength()`** methods modify stream information, while **`addMember()`** and **`removeMember()`** methods add/remove **`RelationContext`** members. These

DOI: 10.1002/spe

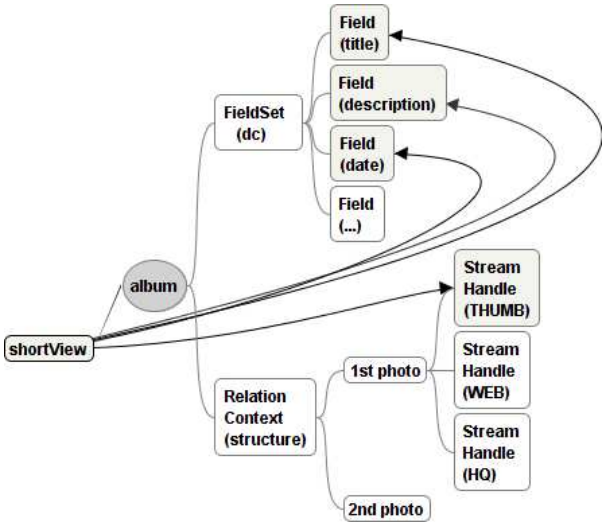
modifications remain buffered unless an explicit *save* is issued by the developer. Saving a DVO is performed by `runtime.saveDO` call, as in `runtime.saveDO(album)` or `runtime.saveDO(book)`. From a developer's perspective, DOLAR's `runtime.getDO` call fetches heterogeneous virtual objects as if the latter originated from a single datastore. The same effective programming metaphor applies when developers store virtual object data, as the `runtime.saveDO` call can store any DVO-based conceptualization to any heterogeneous datastores. DOLAR's `saveDO` uses the *DOStore* interfaces of Figure 7 to ultimately store virtual object information: if the *DVO* driver supports transactions –indicated by `ReadableDOStore.supportsTransactions()` returning a true value– DOLAR uses the `TransactionalDOStore` methods to store virtual object data appropriately. Otherwise, DOLAR uses the `ModifiableDOStore` methods to store virtual object data.

◇ Inserting Data: *Inserting* new items in data collections is a fundamental operation for fostering information expansion. DOLAR virtual objects can be used to insert new items in heterogeneous content stores, using the `getNewDO` and `saveNewDO` Runtime/Dictionary API calls. The former creates a new virtual object, without connecting the object to any storage artifact. For example, the `newAlbum=getNewDO("album")` call creates a new `album` virtual object. After acquiring such a new and "unlinked" virtual object, developers can feed its runtime structures with data using the DVO API. Then, they use the `runtime.saveNewDO` API call to store such an object. The result of this call will be the insertion of a new storage artifact in the datastore furnished as a parameter. For example, the `saveNewDO(newAlbum,"dolar://lib.uoa.gr/pergamos")` call stores the "newAlbum" DVO in Pergamos XML repository by inserting a new "photo-album" XML item.

4.6. Implementation of Composition Schemes



(a) Composition schemes constitute the runtime interface of DVOs



(b) Invoking the shortView composition scheme of a "photo-album" DVO

Figure 8. Composition Schemes

Composition schemes enable us to handle heterogeneous and diverse content items with a uniform DOLAR-runtime interface. Figure 8a shows how schemes designate the interface between business-logic services and DVOs. At specification time, developers define schemes to essentially designate "subsets of a DVO". These consist of any combination of individual *FieldSet*, *Field*, *Stream* and *RelationContext* elements held in a virtual object specification. At runtime, DVOs use such scheme definitions to supply applications with views of corresponding `FieldSet,Field,Stream` and `RelationContext` runtime DVO structures. For example, should we consider the "photo-album" DOP definition of Figure 5, the DOP includes a specification of a `shortView` composition scheme, offering the album's title, description and date along with the thumbnail of the album's first photo (`structure[0].thumb`). At runtime, when the `shortView` scheme is invoked on a "photo-album"

DOI: 10.1002/spe

DVO, the DVO provides its **title, description** and **date Field** runtime structures, accompanied by the **thumb**-named **Stream** of its first child/photo. Figure 8b shows a graphical representation of the **shortView** scheme as executed at runtime, demonstrating the scheme-based exposure of DVO structures.

During instantiation, DOLAR dynamically attaches schemes to corresponding DVOs. The schemes available on a DOP provide the named operations DVOs can respond to, designating the DOLAR-specific interface to these DVOs. Supplying diverse virtual objects with a common set of composition schemes offers service actors a uniform set of virtual object "messages" that hide any data origin, storage or structural details. This is critical, as we seek objects that can be defined by their responses to "messages" and not by their internal representation [20]. Service actors can then use such composition schemes to realize various aspects of application-inherent behavior in a uniform manner, including content presentation, modification, indexing and storage. Fostering DVO usage among varying service provisions, the response to scheme-based "messages" such as the **shortView** of Figure 8b, strictly pertains to exposing a subset of DVO runtime structures. The response does not provide any service-specific transformation or composition of data held in these structures. To this effect, service actors can synthesize the **FieldSet, RelationContext** and **Stream** structures to offer any application-inherent behavior.

4.7. A Comprehensive DVO Usage Example

In this part we demonstrate a comprehensive DVO usage scenario in which a service requires the titles of an **album** and a **book** content items. Initially, the service instantiates the DVOs that correspond to the two items of interest and subsequently, the service fetches the values of their **title** fields, as follows:

```
DVO album = runtime.getDO("dolar://lib.uoa.gr/pergamos/album:100")
DVO book = runtime.getDO("dolar://history.uoa.gr/anthemion/book:12")
aTitle=album.getFieldSet("dc").getField("title").getValue("en")
bTitle=book.getFieldSet("data").getField("title").getValue("en")
```

Album titles originate from **dc:title** metadata values held in XML, while book titles originate from a relational database. The DVOs use their instantiation DOStore drivers to transparently fetch the titles from the respective storage artifacts. For example, the first time the **book** virtual object receives a **getFieldSet("data")** call, it uses its **anthemion** DOStore driver to issue: **anthemion.loadFieldSet("book","12","data")**. Our **anthemion** driver implementation contacts the underlying database and fetch an appropriate SQL query to load the virtual "data"-termed field set:

```
SELECT title, author, ... FROM book WHERE id = 12
```

The **book** virtual object then stages the query return values in individual **Field** structures. Respectively, the **album.getFieldSet("dc")** DVO API call leads the **album** DVO to contact its **pergamos** driver to fetch the values of the given field set: **pergamos.loadFieldSet("album", "100","dc")**. In turn, **pergamos** driver uses the web-service machinery supported by the XML repository to acquire the XML-encoded DC metadata and parse it. The **album** object finally stages individual DC metadata values in respective **Field** structures.

Virtual objects can also automate more complex data fetching operations. For instance, to obtain the thumbnails of the first photo of an **album** and the first page of a **book**, the service issues the following calls:

```
DVO photo=album.getRelationContext("structure").getMember(0)
String photoThumb=photo.getStream("thumb").getReference()
DVO page=book.getRelationContext("structure").getMember(0)
String pageThumb=book.getStream("thumb").getReference()
```

The very first time the **book** DVO receives a **getRelationContext("structure")** request, the DVO uses its **anthemion** driver to load the specific pages: **anthemion. loadRelationMembers("book","12","structure")**. Then, **anthemion** issues the SQL query:

```
SELECT page.id FROM page,book WHERE page.bookId=book.id AND bookId=12
ORDER BY page.order
```

loading the identifiers of the pages of the given book. Respectively, the **album. getRelationContext("structure")** call directs the **album** DVO to contact its **pergamos**

Prepared using speauth.cls
DOI: 10.1002/spe

driver, as in **pergamos.loadRelationMembers("album","100","structure")**, to load the members of its **structure** relationship. The subsequent **getMember(0)** call leads to the provision of a **photo** virtual object; DOLAR uses the identifier of the first photo in the **structure**-termed **RelationContext** to instantiate the respective **photo** DVO automatically and then returns this DVO to the caller. After obtaining the **photo** DVO, the service issues the **photo.getStream("thumb")** DVO API call to get the photo's **thumb**-termed **Stream** structure. The **photo** DVO uses its **pergamos** driver to fetch thumbnail information from the underlying XML repository: **pergamos.loadStreamInfo("photo","photoid","thumb")**. In similar fashion, the **page** DVO uses its **anthemion** driver to fetch thumbnail data via SQL:

```
SELECT jpegUrl,"image/jpeg",jpegLength FROM page WHERE page.id=<pageid>
```

Such a direct DVO API exposure as shown above, is not the prime DOLAR usage pattern. Instead, the strength of the DVO API comes from the composition schemes. In the spirit of the **shortView** "album" scheme of Figures 5 and 8, we can define a common **titleView** scheme on both album and book virtual object specifications; the album's **titleView** offers the album's title and first photo, while and the book's **titleView** offers the book's title and first page respectively. Now, to fetch the title and the thumbnail, the service needs only to fetch the **titleView** scheme on the virtual objects, without engaging any couplings on the structural arrangements of "album" and "book" items:

```
DVO dvo = runtime.getDO(dolarURI)
Scheme titleView = dvo.getScheme("titleView")
String title = titleView.getElement("title").getValue("en")
String thumbURL = titleView.getElement("thumb").getReference()
```

DVO composition schemes project the structure of underlying content items to match the expectations of the business-logic service. This results into service actors exclusively coupled to such scheme-based views and not to any particular data-inherent structural arrangements. Hence, application-inherent compositions of data can be carried out in a uniform coding fashion.

5. DOLAR-BASED SERVICE PROVISION IN PERGAMOS

In this section, we present the DOLAR-based MVC service provision in Pergamos. Figure 9 depicts the DOLAR-based MVC architecture, showing the realization of our content browsing service as an example; all other Pergamos services follow a similar pattern. In particular, our front and back-end users issue HTTP requests with the help of their web-browsers. These requests are processed by our "controller" actors which realize service provision by initially instantiating virtual objects –our "model" actors– and subsequently composing virtual object schemes in terms of our **HTMLEngine** "page template" facilities –our "view" actors. For brevity in Figure 9, we omit *DOStore* drivers which are the "data access" actors.

5.1. Setting up the DOLAR Dictionary

Building a DOLAR-based virtual information space involves straightforward dictionary/domain registration steps. Highlighting the simplicity of this process, the following snippet shows the setup of the **lib.uoa.gr** Pergamos domain of Figure 2a:

```
Dictionary dict = runtime.getDictionary()                                    1
Domain domain = dict.registerDomain("lib.uoa.gr")                            2
DOStore xml = new XMLDOStore("http://RepositoryIP/repo")                     3
domain.registerDOStore(xml, "pergamos")                                      4
DOIndex ft = new FullTextIndex("/opt/pergamos/lucene")                       5
domain.registerDOIndex(ft, "fulltext")                                       6
DOIndex dbIndx = new DCTermsDBIndex("jdbc:mysql://DBServerIP/dc")            7
domain.registerDOIndex(dbIndx, "dc")                                         8
DOPSource src = new FileSystemDOPSource("/opt/pergamos/dops")                9
domain.registerDOPSource(src, "main")                                        10
```

In the above, we assume that the definitions of individual *DOPSource*, *DOStore* and *DOIndex* elements are already in place. Lines 1 and 2 acquire the DOLAR dictionary and register **lib.uoa.gr** as a new domain. Lines 3 and 4 register our **XMLDOStore** driver in the above domain using the **pergamos** identifier. Since the underlying XML repository of Pergamos does

DOI: 10.1002/spe

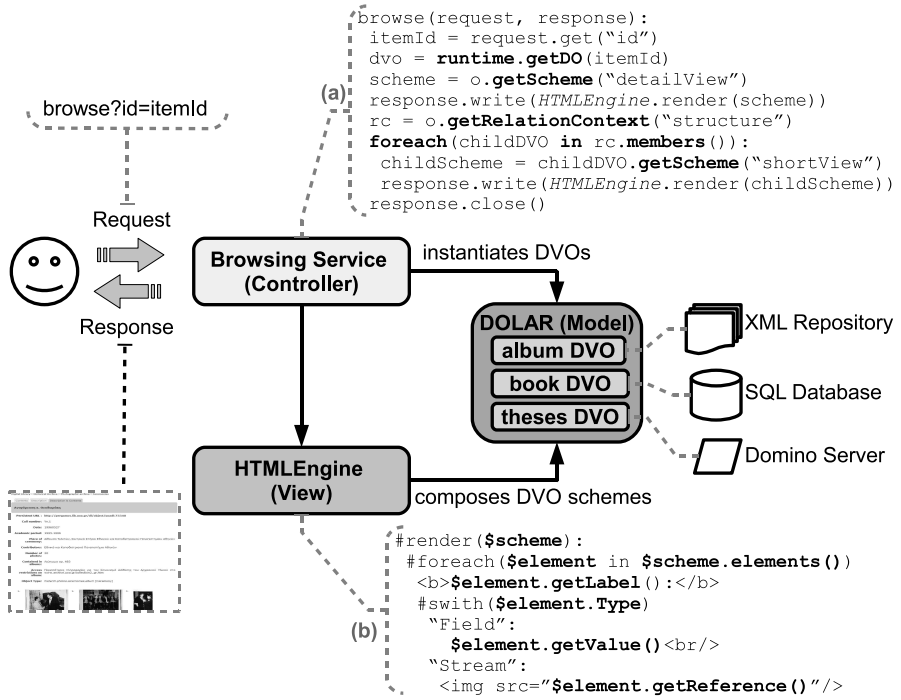


Figure 9. DOLAR-based realization of the MVC architecture in Pergamos

not support transactions, the **XMLDOStore** realizes the *DOStore* mechanism by implementing the **ReadableDOStore** and **ModifiableDOStore** interfaces of Figure 7 as follows:

XMLDOStore **implements** *ReadableDOStore*, *ModifiableDOStore*

The **XMLDOStore** driver implementation wraps Pergamos XML repository and its Web Services residing in the provided HTTP base-URL. Lines 5 and 6 register a **FullTextIndex** with the **lib.uoa.gr** domain, while lines 7 and 8 add our **DCTermsDBIndex** in the dictionary. Consequently, the DOLAR URIs for the two *DOIndex* implementations are **dolar://lib.uoa.gr/index/fulltext** and **dolar://lib.uoa.gr/index/dc** respectively. We discuss these *DOIndex* elements in the next section. Finally, lines 9 and 10 register our **FileSystemDOPSource** with **lib.uoa.gr** domain using the name **main**.

The aforementioned steps may be readily included in an application startup procedure. For instance, these steps can be combined with any initialization actions that the application might require, including processing of configuration settings, establishing database connections and loading of libraries. Also, the dictionary API permits for the dynamic expansion of the virtual information space at runtime.

5.2. Content Presentation

In the content browsing service implementation of Figure 9a, the service accepts user-supplied requests containing the item identifier, as in **browse?id=itemId**. Based on the **itemId**, the browsing "controller" instantiates the DVO that corresponds to the underlying stored item using the **runtime.getDO** DOLAR API call. The controller then uses our **HTMLEngine** "view" actor to render a detailed view of the DVO using the **detailView** scheme. As Figure 9b depicts, the **HTMLEngine** "view" actor composes the structures provided by the **detailView** scheme to transform the data contained in these structures in terms of HTML. In similar fashion, the "controller" proceeds by iterating over the "children" of the DVO. These are provided by the **structure**-termed **RelationContext**. The "controller" offers a short view of the data entailed in such "children" objects. Specifically, for each "child"-item, the "controller" instantiates the respective DVO and then fetches its **shortView** scheme. The scheme is then composed by the **HTMLEngine** to display the short view of the item.

Prepared using speauth.cls
DOI: 10.1002/spe

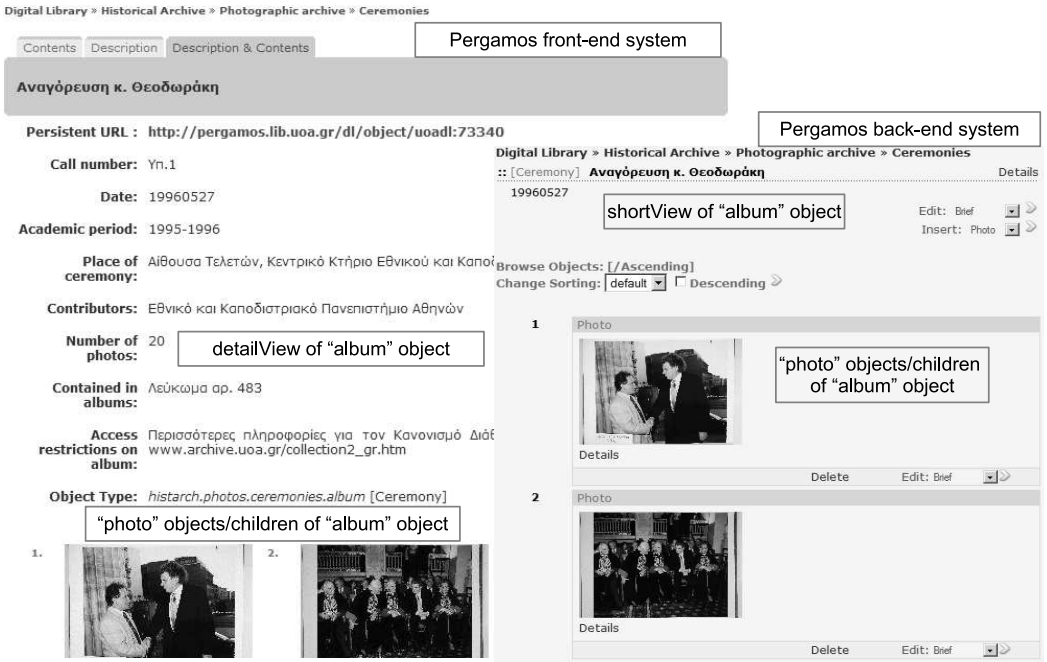


Figure 10. Pergamos Content Browsing Services

Figure 10 depicts the UI of our front and back-end browsing Pergamos services. These services offer a user-consumable hierarchical display of any Pergamos content items in a uniform manner. The services can include any type of items in their HTML display as long as the corresponding virtual objects contain a **`detailView`** and a **`shortView`** composition schemes.

5.3. Content Update Services

Our content update/curation services build upon DOLAR's support of two-way data flows between virtual objects and stored items. In addition, with the use of DVO composition schemes, we automate the generation of 'first-pass" and detailed curation web forms for all types of Pergamos content. Here, we present the implementation of our **`editObject`** content update service which uses DOLAR's **`getDO`** and **`saveDO`** calls. Our **`createObject`** which is used for content insertion is realized in a similar fashion by utilizing DOLAR's **`getNewDO`** and **`saveNewDO`** facilities.

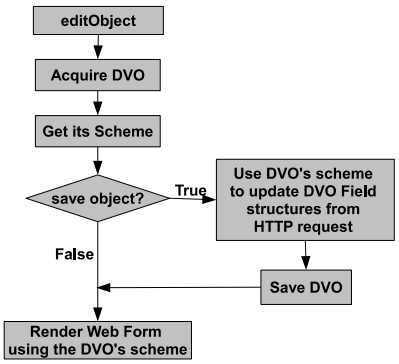
Figure 11a depicts our **`editObject`** service implementation, which uses the **`shortEdit`** and **`detailEdit`** composition schemes to generate web forms for any content items, regardless of the user's language, the data origin or structural details involved. The service updates the underlying stored items, offering the Pergamos content update/curation service of Figure 11b. As the figure shows, the **`editObject`** provides different content curation forms for different types of items in a uniform coding manner. Using the "Save" button, users post their modified form(s) to the **`editObject`** which stores enclosed data. In particular, the **`editObject`** service accepts two forms of HTTP requests:

- **`editObject?id=itemId&short=true|false&lang=langId`**: this request generates a short/detailed web form for a content item in the given language.
- **`editObject?id=itemId&short=true|false&lang=langId&save=true&field1=val1...`**: this request saves the form generated above. The service processes the form's field/value pairs and uses the corresponding DVO identified by **`itemId`** to store form values to the underlying datastores.

Figure 11a shows our **`editObject`** processing input parameters in lines 1–4. Lines 5–9 acquire the user's language along with the virtual object and its **`shortEdit`** or **`detailEdit`** scheme. If the **`save`** parameter is:

DOI: 10.1002/spe

```
1  editObject(request, response)
2   itemId = request.get("id")
3   isShort = request.get("short")
4   save = request.get("save")
5   lang = request.get("lang")
6   dvo = runtime.getDO(itemId)
7   if(isShort.equals("true")):
8    scheme = dvo.getScheme("shortEdit")
9   else:
10   scheme = dvo.getScheme("detailEdit")
11  if(save.equals("true")):
12   foreach(field in scheme.elements()):
13    fieldId = field.getId()
14    fieldValue = request.get(fieldId)
15    field.setValue(lang, fieldValue)
16   runtime.saveDO(dvo)
17   response.write("Success!")
18  end-if
19  response.write(HTMLEngine.renderForm(scheme))
```



a. Pergamos Content Update Service Implementation

b. Pergamos back-end Content Update/Curation Web-forms

Figure 11. Pergamos Content Update Service

- not "true", then the request refers to generating a scheme-based web-form for the item in question. The service proceeds by calling our **`HTMLEngine`** "view" actor to compose the DVO's scheme in terms of web-form input fields in line 19. Our **`HTMLEngine.renderForm()`** facility operates in a similar fashion to **`HTMLEngine.render()`** of Figure 9b. Should we consider that the **`shortEdit`** scheme of our "photo-album" items contains **`callNumber`**, **`title`**, **`date`**, **`period`** and **`place`** **`Fields`**, Figure 11b shows the composition of these **`Field`** structures to offer a user-consumable as well as modifiable view of "photo-album" items.
- "true", then the request intends on saving the generated web-form. Here, the service updates the DVO's **`Field`** structures with the user-supplied values using the DVO's scheme in lines 12-15. It then stores the DVO using the DOLAR **`runtime.saveDO(dvo)`** call of line 16 and re-renders the just-updated fields of the web-form using the **`HTMLEngine`** in line 19.

DOLAR offers a uniform solution in terms of coding the **`editObject`** service, regardless of any data-inherent structural and storage details. The service can generate short and detailed web forms for any items that provide **`shortEdit`** and **`detailEdit`** composition schemes.

6. INDEXING & SEARCHING VIRTUAL OBJECTS

In operational environments, applications use heterogeneous indexing/searching options to offer users different types of search functionality. In this section, we focus on the ability of DOLAR to automatically index new types of content and also display such new content in search results. Specifically, as the information space expands, our approach automates the process of: (a) indexing new content items in existing index facilities and (b) including new content items in existing

Prepared using speauth.cls
DOI: 10.1002/spe

provisions of search results. Our *DOIndex* API views index values, possibly composite as well, as stored projections of DVO-entailed data. The API provides the following two indexing methods:

- **`DOIndex.addOrUpdate(objectId,dopId,scheme):void`** – adds or updates an index entry, identified by **`objectId`**. The entry's data originate from the given DVO composition scheme.
- **`DOIndex.addOrUpdate(objectId,dopId,DVO):void`** – adds or updates an index entry, obtaining data from the provided DVO.

The above two methods designate the "indexing" behavior of a *DOIndex*; implementations may operate atop heterogeneous indexing options such as databases, full-text search engines or RDF triple stores. As the parameters of the methods show, a *DOIndex*-based record always includes the object identifier (**`objectID`**) which is required by DOLAR to instantiate virtual objects. We also include the DOP identifiers of DVOs (**`dopId`**) to help users limit search results in terms of specific conceptualizations. In Pergamos, the use of such DOP identifiers allows users to search for specific types of items, such as "photo-albums" or "books". The first variation of the **`addOrUpdate`** method uses the provided composition scheme as the source of data to be indexed, while the second offers a full exposure of DVO data to the *DOIndex* implementation. This proves useful in cases where applications need to index entire DVO data.

Our *DOIndex* API also provides the following search-wrapping operations:

- **`DOIndex.search(String):String[]`** – returns the indexed items that match the criteria involved in the given query, expressed as a string.
- **`DOIndex.search(String,start,count):String[]`** – returns **`count`**-numbered items that match the given string query, starting from **`start`**.
- **`DOIndex.countObjects(String):long`** – returns the number of indexed items that match the given string query. Applications can combine this method with the above one to offer "pagination" of search results.

These three search methods provide search results in terms of the matching items' identifiers only, regardless of the information that is indexed beneath or the query mechanism supported. Search services use such identifiers to instantiate respective DVOs and then utilize DVO composition schemes to display search results to the end-user. The methods outline the "search" behavior of a *DOIndex*; implementations may support their own query language or any other search syntax of choice. For example, should a *DOIndex* implementation use a relational database to index data, the implementation in question will apparently use SQL to pose search queries.

Our goal in creating *DOIndex* is to offer a thin layer or "*adapter*" mechanism between the application logic and any underlying indexing/searching facilities. Applications implement the *DOIndex* API methods to exploit DOLAR information expansion benefits. Here, we show that the *DOIndex* mechanism in combination with composition schemes, enables developers to avoid the crosscutting of content indexing/searching concerns in service provision code. This ultimately offers extensible implementation of service provision actors in terms of indexing/searching too, allowing applications to support newly-introduced types of items without any modifications. For brevity, we focus our presentation on the scheme-based variation of the **`addOrUpdate`** method and outline its usage in Pergamos, discussing the realization of our two search services.

6.1. Achieving Indexing of Content in a Uniform Manner

When users modify data, services have to follow suit in modifying application indexing facilities accordingly. The issue here is to perform such updates without coupling service implementations to specific content types or indexing options. In Pergamos, we support two search facilities namely, a full-text index wrapped by our **`FullTextIndex`** implementation and a relational database wrapped by our **`DCTermsDBIndex`**. The former operates on **`fullTextRecord(objectId,dopId,text)`**, while the latter on records of DC terms: **`dcRecord(objectId,dopId,title,date,creator,contributor,description,subject,coverage`**). We use **`DCTermsDBIndex`** to offer a mapping between: (a) the diverse kinds of fields employed by our content items and (b) DC-based metadata fields. For instance, consider the case that the **`editObject`** service of Section 5.3 needs to index "folklore-artifact", "album-of-photos" and "book-of-pages" items. Here, "book" and "album" items carry text-based data, "photo" and "page" items maintain image-based digital content, and finally,

Prepared using speauth.cls
DOI: 10.1002/spe

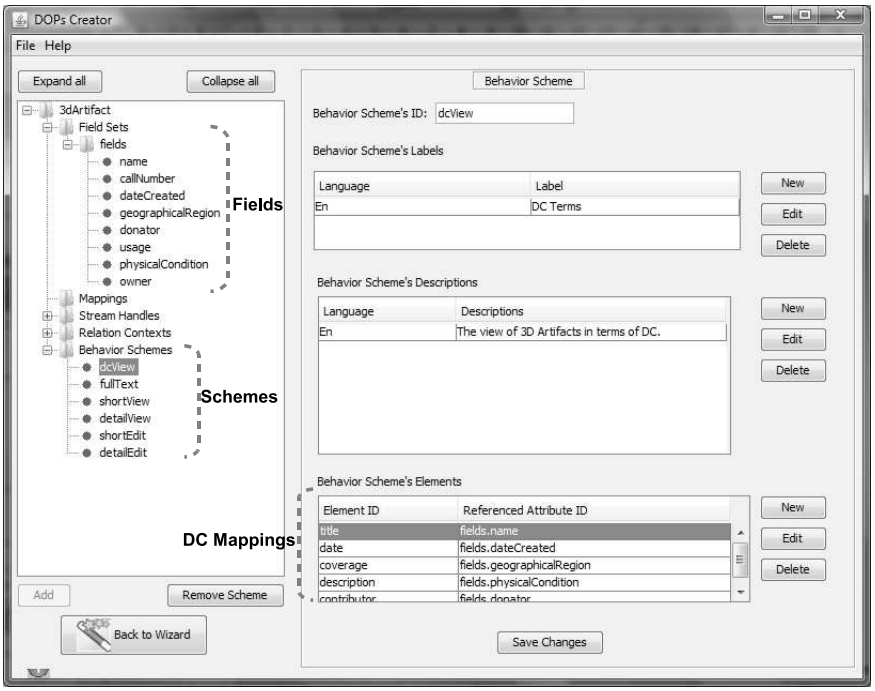


Figure 12. Using DOPs Creator to Define Scheme-based Mappings

"folklore-artifact" items carry both text and image data. In this case, we need to avoid crosscutting of indexing concerns in our **`editObject`** service. In particular, the service has to:

- index any items in a *uniform* manner, regardless of the items' idiosyncrasies or the particular details of the underlying indexing facilities.
- avoid indexing non-appropriate items. For example, the service should not populate the indexes with non-text entries for our "page" and "photo" items, as such entries are not usable by text-based searches.

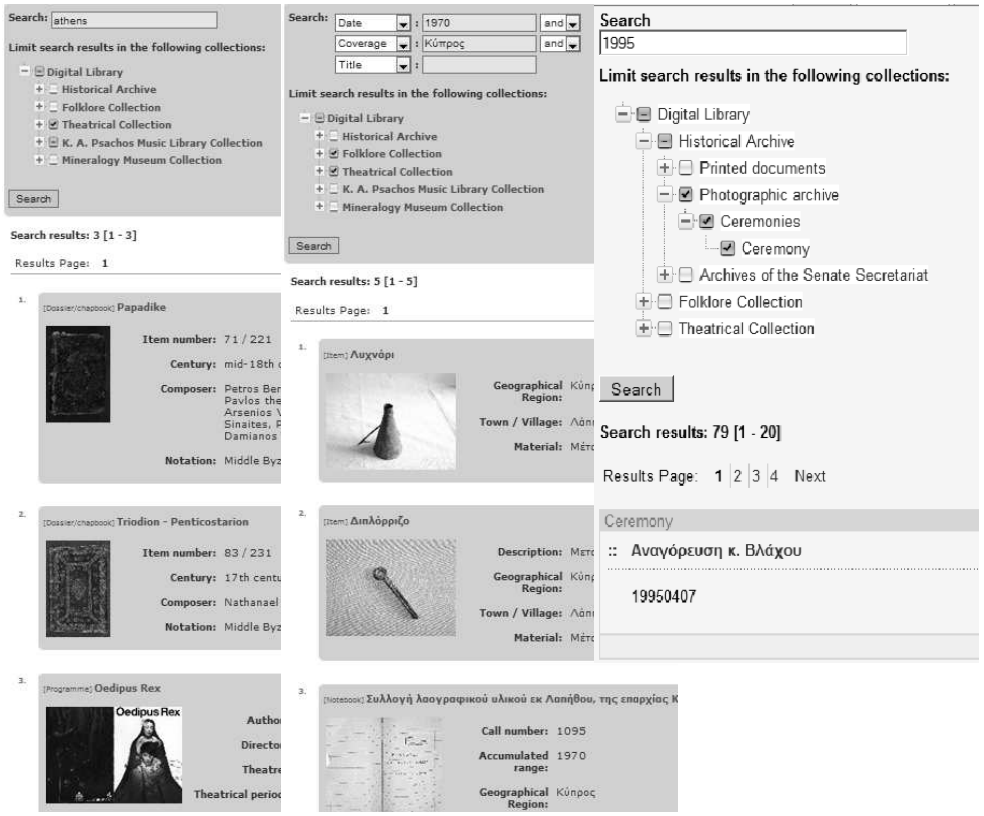
Dealing with these two issues will allow **`editObject`** to index new types of items without code modifications. To address these issues, we connect a *DOIndex* implementation to a specific composition scheme. This connection is established by the **`DOIndex.getIndexScheme()`** method that provides the name of the scheme supported by the given *DOIndex* implementation. For example, our **`FullTextIndex`** "knows" how to index DVOs that offer a **`fulltext`**-termed scheme. Such a scheme, when present in a DVO, provides the particular **`Field`** structures that must be full-text indexed. Developers decide which types of content should be included in the full-text index and issue corresponding **`fullText`**-termed schemes. Similarly, our **`DCTermsDBIndex`** "knows" how to index DVOs that contain a **`dcView`**-termed composition scheme. When present in a DVO, the **`dcView`** scheme offers a mapping between the fields of a virtual object and DC fields. Figure 12 depicts the definition of the 'folklore-artifact" **`dcView`** scheme with the help of our *DOPs Creator*.

Right after storing a DVO, our **`editObject`** service calls our **`updateIndexes`** facility, updating Pergamos indexes as follows:

```
updateIndexes(dvo):                                                    1
 libDomain = runtime.getDictionary().getDomain("lib.uoa.gr")           2
 foreach(index in libDomain.registeredIndexes()):                      3
  schemeId = index.getIndexationScheme()                               4
  scheme = dvo.getScheme(schemeId)                                     5
  if (scheme != null):                                                 6
   index.addOrUpdate(dvo.getURI(), dvo.getDOP().getURI(), scheme)      7
```

Line 2 acquires the **`lib.uoa.gr`** domain of the DOLAR dictionary. As discussed in Section 3, this domain contains our two **`FullTextIndex`** and **`DCTermsDBIndex`** *DOIndex* elements, supplying services with pointers to any Pergamos-pertinent indexing facilities. Line 3 iterates through any *DOIndex* implementations registered in the domain using the **`Domain.registeredIndexes()`**

DOI: 10.1002/spe



(a) Pergamos front-end and back-end search services

```
1 search(request, response):
2  text = request.get("text")
3  dopIds = request.get("dops")
4  query = constructQuery(text, dopIds)
5  ftIndex = libDomain.getDOIndex("fullText")
6  ids[] = ftIndex.search(query)
7  foreach(id in ids):
8   dvo = runtime.getDO(id)
9   sch = dvo.getScheme("shortView")
10  response.write(HTMLEngine.render(sch))
```

(b) Full-text search service implementation

Figure 13. Various aspects of Pergamos Search Services

DOLAR API call. For each *DOIndex* obtained, line 4 fetches the name of the scheme supported, using the **`getIndexScheme()`** *DOIndex* API call. Line 5 then uses this name to fetch the respective composition scheme from the newly stored DVO. If the DVO contains such a scheme, line 7 issues the *DOIndex* API **`addOrUpdate`** call to index DVO data, supplying the DVO URI, its DOP identifier along with the particular scheme acquired in line 5. The presence of a particular composition scheme on a particular DVO indicates whether such a virtual object should be included in a given index. This way, the indexing concerns are effectively separated and the knowledge about: (a) "what to index" is represented by a composition scheme, (b) "where to index" is represented by a *DOIndex* DOLAR URI and (c) "how to index" is represented by a *DOIndex* implementation.

6.2. Searching with the DOIndex Mechanism

Figure 13a shows various aspects of our back- and front-end Pergamos search services, utilizing our full-text and DC-based indexing facilities. Our search services allow users to limit search results in the digital collection hierarchy as far as types of content items is concerned. These types are distinguished through their respective DOP identifiers. Our free-text search service employs the **`DOIndex.search`** method of **`FullTextIndex`** to fetch free-text queries; in this, **`DOIndex.search`** uses the underlying Lucene full-text engine's query syntax as in:

```
text:'athens' and dopId:'album'
```


Prepared using speauth.cls
DOI: 10.1002/spe

Respectively, our field-based search uses the **`search`** method of our **`DCTermsDBIndex`** to fetch SQL queries via the underlying DC records database, as in:

```
SELECT objectId FROM dcRecord WHERE dopId="3dArtifact" AND date="1970"
```

Figure 13b shows our full-text search service implementation; the DC-oriented search is realized in a similar fashion. As the figure shows, our search services process user input (lines 2,3), constructing a corresponding query (line 4). Then, line 5 acquires the particular *DOIndex* from the **`lib.uoa.gr`** domain. For example, the full-text search service will acquire the **`FullTextIndex`**, while the field-search will obtain the **`DCTermsDBIndex`**. Line 6 then invokes the **`DOIndex.search`** method to fetch the query via the underlying indexing facility (line 6). Finally, search services use the DOLAR URIs returned by the **`search`** method to instantiate DVOs and then employ the **`shortView`** composition scheme to render the display of search results in terms of the **`HTMLEngine`** facility (lines 7-10). In this lineup, our search services do not engage in couplings to any particular types of content items fostering extensibility of the application.

As we have shown, search services use a *DOIndex*-based search to fetch DOLAR URIs of search results. Moreover, browsing services use **`RelationContext`**-based relationships to fetch the DOLAR URIs of content items. Both our services synthesize DVO schemes to feed the user-display in a uniform manner regardless of any idiosyncrasies of contributing content items.

7. EVALUATION

In this section, we evaluate the effectiveness and performance of our approach, showing:

- how DOLAR meets the challenging requirement to gradually expand the information space with new types of content, without modifying any service provision actors. In Pergamos, we show how the use of DOLAR exploits MVC benefits, yet, without requiring to issue a different MVC realization for each different type of content.
- that DOLAR-imposed overheads are not significant in terms of performance and DOLAR-based service implementations scale as well as directly-coded implementations atop both SQL and XML data-sources.

7.1. Effective Information Space Expansion

As mentioned in Section 1, when new items join the information space, applications need to effectively deal with the following cases:

(1) support newly-encountered types of data sources: Since the application has to firstly access the items for them to partake in the service provision, the extension of the "data access" actors supported by the application at hand cannot be avoided. In Pergamos, for example, we need to include items that originate from remote collections such as the Domino-based theses or database books. In order to support a novel data source, the "expanding" application may have to revise its "data access" actors or even introduce new ones. The critical issue here is how expensive is to perform such "data access" extensions.

(2) support novel types of collections: In Pergamos, for example, new digitization projects emerge, introducing new types of items that need to be created with the help of the Pergamos documentation services and then be inserted in existing Pergamos datastore(s). Here, the fundamental issue is how flexible is the underlying datastore(s) to support such novel collections of content.

(3) include new items in existing services: in both of the above cases, the key issue is to enable existing business-logic to deal with new content items without "breaking" its implementation.

To deal with case (1) and (2), our approach uses the *DOStore* mechanism. To support a novel type of datastore, developers have to define a new *DOStore* driver, while to support a novel collection, they may need to revise an existing driver. The main value addition of DOLAR is in dealing with case (3), as developers use the DOPs Creator tool to issue virtual object specifications and subsequently, they use composition schemes to supply virtual objects with service-compatible interfaces. This way,

Prepared using speauth.cls
DOI: 10.1002/spe

on one hand, developers avoid the manual coding of new business-logic objects. On the other, the use of composition schemes helps them avoid the revision of services. To this effect, applications can deal with the expansion of the information space without requiring any business-logic code modifications.

To show the effectiveness of our approach, we use the following Pergamos examples, where the information space expands with (a) the addition of the *Anthemion* database collection of books and (b) the digitization of the *Byzantine Music Manuscripts*. In what follows, we succinctly outline our findings and/or experience when our approach deals with the aforementioned cases (1), (2) and (3).

(1) Supporting new types of datastores: To support the Anthemion database, developers have to define a new *DOStore* driver by realizing the **`DOStore`** API interfaces of Figure 7. In particular, developers realize the **`ReadableDOStore`** interface so that DOLAR may gain read-only access to a datastore. In addition, to support data modification, developers realize either the **`ModifiableDOStore`** or the **`TransactionalDOStore`** interface, depending on whether the underlying datastore supports transactions. Provided that the relational database underlying the Anthemion collection of books supports transactions, the new *DOStore* driver is defined as:

`AnthemStore` **implements** *DOStore*, *TransactionalDOStore*

The realization of the *DOStore* API interfaces is straightforward; our experience indicates that it invariably takes a short period of time to define a new *DOStore* driver. For example, to allow DOLAR to fetch the data of the Anthemion database, it practically requires the implementation of three methods, namely, the **`ReadableDOStore.loadFieldSet()`**, **`ReadableDOStore.loadRelationMembers()`** and **`ReadableDOStore.loadStreamInfo()`**. It is worth pointing out that the realization of the remaining **`ReadableDOStore`** interface methods is trivial. Furthermore, to enable the support of data modification, the developer has to essentially realize five methods of the **`TransactionalDOStore`** or the **`ModifiableDOStore`** interfaces, namely, the **`saveFieldSet()`**, **`saveRelationMembers()`**, **`saveStreamInfo()`**, **`deleteObject()`** and the two **`addNewObject()`** variants. Table I shows a high-level description of the definition of the **`AnthemStore`** driver, reflecting the simplicity of the *DOStore* driver definition process.

After defining the new *DOStore* driver, developers register it with the DOLAR dictionary using the following:

```
Dictionary dict=runtime.getDictionary()
Domain domain2=dict.registerDomain("history.uoa.gr")
DOStore anthem=new AnthemStore("jdbc:oracle:thin@//IP/anthem")
domain2.registerDOStore(anthem, "anthemion")
```

The extension of the virtual information space is performed via simple registration steps, advancing automation. In the first two lines of the above snippet, we register the **`history.uoa.gr`** domain in the DOLAR dictionary. In the next two, we register our **`AnthemStore`** driver using the **`anthemion`** identifier; this yields the fully qualified **`dolar://history.uoa.gr/anthemion`** URI for the new DOStore driver which wraps the *Anthemion* Oracle database using the JDBC library.

The addition of new "data access" actors is performed without "breaking" the application, due to the effective separation of information contexts offered by the virtual information space. For instance, the addition of the **`history.uoa.gr`** domain and its respective *DOStore* driver does not interfere with the **`lib.uoa.gr`** domain and vice versa.

(2) Supporting novel types of content: The introduction of a new type of content in a datastore already registered with DOLAR predominantly depends on the features and the flexibility of this datastore. For example, to introduce a new type of items in the custom-made *Anthemion* "book" database necessitates the modification of the underlying database schema. Provided that an existing datastore has to change to support a novel type of items, it is evident that the corresponding "data access" actor will have to be changed too. Clearly, this is not a DOLAR limitation. For example, in less-rigid datastores such as XML repositories, the introduction of new types of items may not impose any modifications to underlying XML arrangements and thus, such repositories can better cope with the introduction of new collections. In Pergamos, the internal Fedora XML repository can hold varying types of items in a unified XML datastream storage model [21]. This XML storage model or others such as METS [22], for instance, can encode various content conceptualizations

DOI: 10.1002/spe

ReadableDOStore Interface	
isReadOnly():	Returns **false**.
supportsTransactions():	Returns **true**.
objectIdentifiers():	Fetches SQL query to provide the record identifiers.
objectCount():	Fetches SQL query to provide the number of records.
loadFieldSet():	Fetches SQL query to load the fields of the given "book".
loadRelationMembers():	Fetches SQL query to load the "pages" of the given "book".
loadStreamInfo():	Fetches SQL query to load the information of the given "page".
TransactionalDOStore Interface	
beginTransaction():	Provides a long value for uniquely identifying a new transaction. A data structure is created to hold the SQL statements of the new transaction.
commit():	Commits the transaction identified by the given long value by fetching the SQL statements attached to the internal data structure. Performs a rollback in case of error, indicating the error condition.
addNewObject():	Constructs the SQL statement(s) for inserting a new "book" or "page" and adds them in the transaction data structure.
deleteObject():	Constructs the SQL statements(s) for deleting the given "book" or "page" and adds them in the transaction data structure.
saveFieldSet():	Constructs the SQL statement(s) for inserting/updating the given "book" and adds them in the transaction data structure.
saveRelationMembers():	Constructs the SQL statement(s) for inserting/updating the given "pages" and adds them in the transaction data structure.
saveStreamInfo():	Constructs the SQL statement(s) for inserting/updating the information of the given "page" and adds them in the transaction data structure

Table I. The definition of the AnthemStore driver

in a uniform manner, enabling applications to support novel types of items without changing the underlying data storage "schema". In Pergamos, the use of a flexible XML storage model helped us avoid revising the **pergamos** "data access" actor each time a new collection development project emerged. To this effect, the development of the *Byzantine Music Manuscripts* or any other digitized collection imposes no modifications to our **pergamos** *DOStore* driver.

In general, content stores and respective conceptualizations share a "many-to-many" relationship. Indeed, a given datastore may hold items that abide to multiple conceptualizations, while a given conceptualization may be stored effectively by heterogeneous datastores. To show how our approach can effectively separate the information access from the information conceptualization dimension, Figure 14 presents the SQL schema of our default DOLAR database. Such a database can hold any DOLAR-based conceptualization without changes and it serves as a valuable tool for rapid prototyping of DOLAR-based services and for testing purposes. In Pergamos, we use this database in the testing installation of the system as follows. We create a copy of the DOLAR dictionary we have in place in the production system, including the three domains discussed in Section 3. Each *DOStore* driver of the production DOLAR space is realized as a different instance of the *DOStore* driver of the default DOLAR database. For instance, the **dolar://lib.uoa.gr/pergamos** driver of the production DOLAR space is realized as a driver operating atop the default DOLAR database in the testing Pergamos installation. This configuration helps us test and verify any new services before deploying them in the production system. It also helps us verify the addition of new types of items, in terms of testing new virtual object specifications and their composition schemes. At the end of the day, the virtual object specifications defined in the testing Pergamos installation are deployed to the production system and automatically operate atop the production data-sources.

Our approach dissociates business-logic conceptualizations from any storage-specific details. When the underlying datastore has to change to cope with a novel type of items, the modifications

Prepared using speauth.cls
DOI: 10.1002/spe

```
objects(id, dopid, cDate, mDate)
fields(objects_id, fieldset_id, field_id, lang_id, value)
relationcontexts(objects_id, relctx_id)
relationmembers(objId, relctx_id, memberObjectId)
streamhandles(objects_id, strhandle_id, mime, length, uri)
```

Figure 14. The SQL schema of the default DOLAR database

involved in bringing the respective *DOStore* driver up to date do not interfere with any other component of the application.

(3) Effective inclusion of new items in existing services: Once the "data access" actors are in place, our approach simplifies the inclusion of their items in service provisions. In particular, our approach:

1. provides freedom from directly coding the business-logic objects. Using our DOPs Creator tool, developers issue virtual object specifications and DOLAR processes such definitions to offer business-logic objects automatically. For example, we use DOPs Creator to issue the new "book-of-pages" and "byzantine manuscript" conceptualizations and then store them in terms of an XML DOP definition in our **`FileSystemDOPSource`**.
2. allows developers to make newly introduced business-logic objects compatible with existing services with the help of composition schemes. Schemes designate the runtime interface of virtual objects and services rely on the presence of composition schemes to realize application logic. Consequently, to include any newly added types of items in existing service provisions, the only requirement is to provide proper definitions for the corresponding composition schemes. This is performed during the construction of the virtual object specification with the help of the DOPs Creator, offering effectiveness and automation.

In Pergamos, six composition schemes are used by services, namely, the **`shortView`**, **`detailView`**, **`shortEdit`**, **`detailEdit`**, **`fulltext`** and **`dcView`** schemes. Each such scheme reflects a particular composition issued by Pergamos services:

- **`shortView`** and **`detailView`**: provide the "short" and "detail" view of an item respectively. These schemes are used by content browsing, presentation and search services.
- **`shortEdit`** and **`detailEdit`**: provide the "first-pass" and "full-record" view for editing an item respectively. These schemes are synthesized by content insert/update services.
- **`fulltext`** and **`dcView`**: the first provides the fields of an item to be full-text indexed, while the latter provides the fields of an item that map to DC terms. These schemes are used by content indexing services.

For example, right after registering the new **`AnthemStore`** driver and adding the new "book-of-pages" DOP definition, Pergamos services deal with "book" and "page" virtual objects without modifications in their code. The "book" virtual object specification provides definitions for all aforementioned schemes, while the "page" specification provides only **`shortView`**, **`detailView`**, **`shortEdit`** and **`detailEdit`** schemes, as "page" objects are not included in text-based indexes. When the **`AnthemStore`** joins the information space, we reuse our **`updateIndexes`** facility discussed in Section 6 to index the items that originate from the new store in our full-text and DC-based search facilities:

```
String[] ids = store.objectIdentifiers()
foreach(id in ids):
 String fullId = store.getURI() + "/" + id
 DVO o = runtime.getDO(fullId)
 updateIndices(dvo)
```

Line 1 uses the **`DOStore`** API to obtain the identifiers of the items entailed in a given store. Then, line 2 iterates through these identifiers to add corresponding items in our Pergamos-pertinent indexes. In particular, it creates the DOLAR fully-qualified URI for each content item and then instantiates the corresponding DVO. Finally, in the last line it calls our **`updateIndexes`** facility, indexing the DVO in our **`FullTextIndex`** and **`DCTermsDBIndex`** of Section 6. This way, Pergamos can index the "book" items in the full-text and DC indexes, while ignoring non-textual "page" items. The use of composition schemes offers a uniform way to include newly added content in existing Pergamos indexes, contributing to the automation of the expansion process. When a new datastore

DOI: 10.1002/spe

joins the information space, composition schemes automate the "discovery" of new items, allowing Pergamos to automatically index their items. Similar effectiveness and automation is provided to the other Pergamos services too. Specifically, the browsing service of Figure 9 can automatically browse "book" of "pages", while the content update service of Figure 11 can automatically generate proper web forms for the new "book" and "page" items. Finally, the search services of Figure 13 can automatically include the newly added "books" in their search results.

As services exclusively rely on composition schemes to realize application logic, they avoid engaging any couplings to any particular content origin, storage or structural arrangements. Consequently, new types of content can join the service provision without modifying any service actor implementations.

7.2. Experimental Evaluation

Here, we present our experiments for measuring the performance of DOLAR in terms of the first "logical" MVC task, which is the staging of data in business-logic "model" objects. The second "logical" MVC task, which is the transformation/presentation of the data, is not being measured, as it would be identical in all cases (it would refer to same synthesis of data performed on behalf of the Pergamos "view" MVC components). It is clear that directly-coded "model" objects which are tailored to the specific datastore offer the best performance. Thus, in our experiments, we compare the throughput –number of items fetched per second– achieved by using virtual objects as business-logic "model" actors against the throughput reached by directly-coded business-logic objects. We use a variety of SQL-based and XML-based data sources and also use a common conceptualization, where the data items consist of the standard DC fields [19]. These data items originate from Pergamos **`dcRecord`** items, discussed in Section 6.

•DOLAR operating atop SQL-based data sources: our dataset includes 100,000 DC items stored as tuples in a MySQL database. We issue three different implementations of "DCItem" business-logic objects and respective "data access" actors:

1. *Directly coded Java/SQL*: We issue a plain "DCItem" Java object, realizing a simple, directly coded Java-based "model" actor. Such a Java object abides to the JavaBeans specification [23] and offers a pair of getter/setter methods for each individual DC field, such as **`getTitle`** and **`setTitle`**. This JavaBean component accesses our SQL-based DC items using a simple **`sqlFetch`** "data access" actor implementation that builds upon the MySQL JDBC machinery.
2. *DOLAR operating atop SQL*: We issue a "DCItem" virtual object specification entailing the standard DC fields in a single *FieldSet* definition. We also define a simple read-only **`sqlTest`** *DOStore* driver implementing the **`ReadableDOStore`** interface of Figure 7. The **`loadFieldSet()`** method of the **`sqlTest`** and the above **`sqlFetch`** facility share an identical SQL fetching code.
3. *Hibernate/SQL*: We also use Hibernate Object-Relational Mapping library to map the "DCItem" JavaBean to the underlying DC-item database table. In this case, SQL fetching is managed by Hibernate and we only provide a Hibernate mapping definition [24].

Figure 15 shows the results of our SQL-based experiments, comparing the amount of time –in milliseconds– required to fetch various amounts of unique DC items in a range of [10,000 - 100,000] items. To capture the worst case scenario, we used random distributions of item identifiers, employing no data caching facility. All executions were performed in the same machine, using the same hardware and software choices. Finally, for each different amount of items, we repeated the experiment ten times and used the average execution time of these ten iterations to generate Figure 15. As the figure shows, in terms of throughput, DOLAR operating atop SQL scales almost as well as directly coded Java/SQL. In particular, directly coded Java/SQL reached an average throughput of 3,688 items/sec, while the use of DOLAR imposed a 20% performance overhead, offering an average throughput of 2,944 items/sec. DOLAR outperforms Hibernate, as the latter

Prepared using speauth.cls
DOI: 10.1002/spe

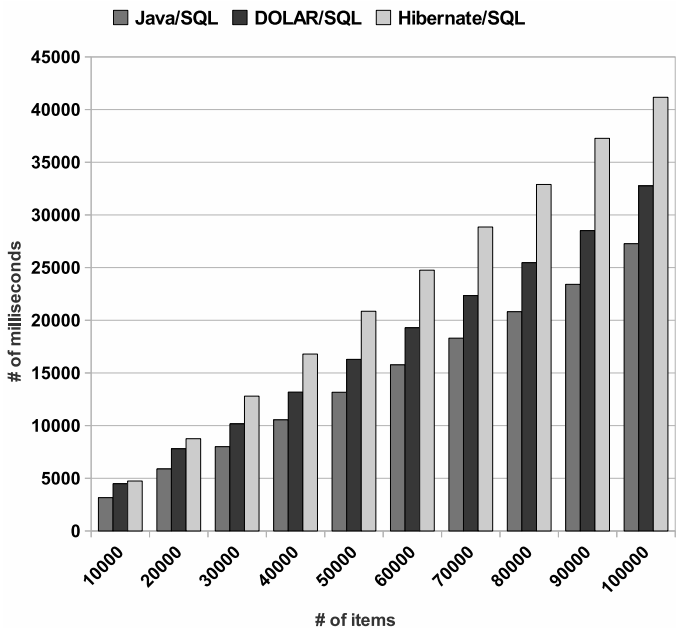


Figure 15. Performance Evaluation of: directly-coded Java/SQL, DOLAR operating atop SQL and Hibernate-based Java/SQL

decreases directly coded Java/SQL performance by 36%, offering an average throughput of 2,364 items/sec.

•DOLAR operating atop XML-based data sources: we compare DOLAR virtual objects operating atop XML-based DC items against directly coded Java/XML approaches. We store our DC items in terms of XML, in a fashion which abides to the "simpledc" XML schema [25]. In particular, the configurations compared are the following:

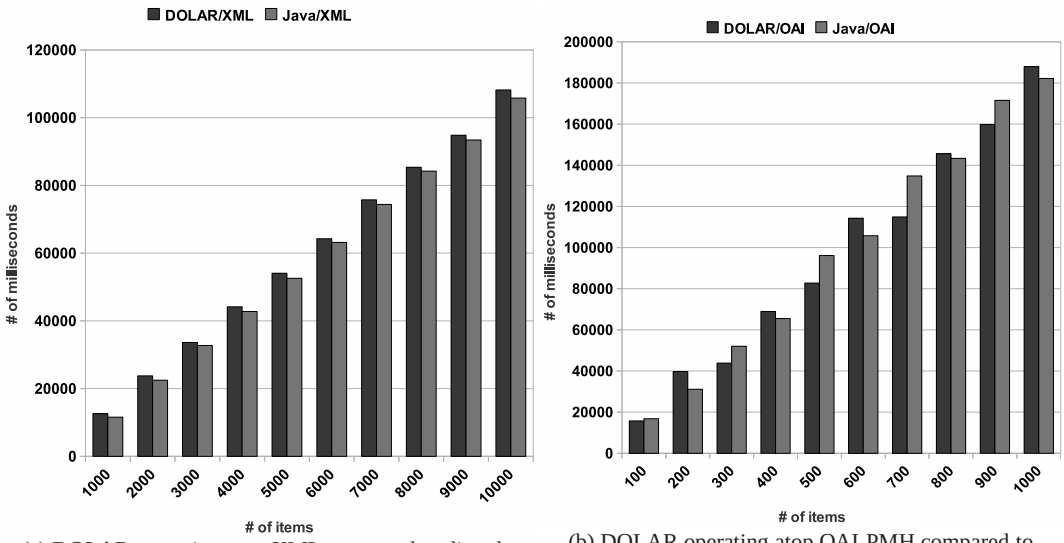
1. *Directly coded Java/XML*: We use the "DCItem" JavaBean mentioned in the SQL experiments to offer a Java-based DC item "model" actor. The directly-coded JavaBean component accesses XML-encoded DC items using a simple **`xmlFetch`** "data access" actor which builds upon default Java XML libraries.
2. *DOLAR operating atop XML*: We reuse the "DCItem" virtual object specification to offer a DVO-based "model" actor. At the same time, we realize a simple **`xmlTest`** *DOStore* driver as a "data access" actor. The **`loadFieldSet()`** method of **`xmlTest`** and the aforementioned **`xmlFetch`** facility share identical XML handling code.

Figure 16a provides the results of our XML-based experiments, comparing (a) the time required to fetch various amounts of unique DC items using **`xmlFetch`** and stage such items using "DCItem" JavaBeans, against (b) the time required to instantiate identical amounts of "DCItem" virtual objects, fetching their **`DC FieldSet`** via the **`xmlTest`** DOStore. As the figure shows, the use of DOLAR imposed no discernible performance overhead: in a range of [1,000 - 10,000] items, the directly coded Java/XML approach offered an average throughput of 93 items/sec, while DOLAR reached an average throughput of 90 items/sec (overhead: 3,06%).

Finally, we illustrate that DOLAR can be used atop any data source, directly reflecting the particular datastore capabilities as well as limitations. Figure 16b presents the results of using DOLAR in an OAI-PMH context. OAI-PMH is an XML-based interoperability protocol, offering a Web-service for metadata harvesting [26]. Here, we fetch XML DC records over the web, directly using our "live" Pergamos OAI-PMH Web-service. This is reflected in the figure where obtained rates do not follow a predictably consistent pattern; the figure compares the throughput of:

1. *Directly coded Java/OAI-PMH*: we reuse the "DCItem" JavaBean as a "model" actor, while we realize a simple **`oaiFetch`** "data access" actor. The latter fetches XML-based DC records

DOI: 10.1002/spe



(a) DOLAR operating atop XML compared to directly coded Java/XML

(b) DOLAR operating atop OAI-PMH compared to directly coded Java/OAI-PMH

Figure 16. Performance Evaluation of DOLAR operating atop XML-based data sources

over the web, using our Pergamos OAI-PMH web service. XML processing is once again performed using default Java XML libraries.

2. *DOLAR operating atop OAI-PMH*: we reuse the "DCItem" virtual object specification, while we realize a simple *testOAI* DOStore driver. The latter shares an identical XML/OAI-PMH processing code with the **`oaiFetch`** facility.

As Figure 16b shows, the use of DOLAR imposes no obvious performance overhead, as both directly-coded Java/OAI-PMH and DOLAR-based OAI-PMH implementations reach an average throughput of 5,5 items/sec. Once again, DOLAR scales as well as the underlying store does.

8. RELATED WORK

There have been a number of approaches for extending programming language mechanisms to accommodate separation of concerns [10]. Such efforts include aspect-oriented programming (AOP) [18, 27], composition filters [28] and hyperslices [29, 30]. Our DOLAR approach separates the four concerns of Figure 1 and this is the extent of the relationship to the aforementioned approaches, as DOLAR neither alters the underlying programming language nor provides Java extensions, as is the case in [31].

DOLAR essentially offers an infrastructure and does not provide a full-fledged independent application. In this context, our proposal does not modify the componentization of an application as the MVC architecture use-case demonstrates. Viewing digital libraries as its primary applications, DOLAR can be used in conjunction with any DL-architecture such as [32, 33, 34, 35]. Our approach is also aligned with the well-documented long-term objective to offer a unified foundation for digital libraries; this objective has been articulated in a number of efforts including the definition of digital object repositories [36], the 5S formal model of digital libraries [37], the formal model for annotating digital content [38] and the OAIS [39] and DELOS [40] reference models.

XML-based approaches can separate data from presentation, a significant requirement for expanding information spaces. For example, XML approaches may use RDF [41] to issue business-logic conceptualizations and may simultaneously employ XSLT [42] to transform XML data for presentation. Should we employ XML-based conceptualizations in Pergamos, they apparently add value to our front-end "store-to-user" services. However, in the reverse "user-to-store" flow, where the information originates from the user to be stored in an underlying datastore, the use of XML

Prepared using speauth.cls
DOI: 10.1002/spe

adds value only if: (a) the users supply data in XML-based formats, or (b) the data is stored in XML-based formats. In operational settings that do not comply with either of the two conditions, the use of XML fails to add value in the realization of "user-to-store" information flow. For example, having the Pergamos back-end subsystem process a user-supplied web-form to yield XML representations of data adds no value in the case of "book" items, since "book" information is to be stored in the relational database that handles *Anthemion* book items. Hence, to support such "user-to-store" flows, applications have to employ an additional set of operations to manage storing data in underlying non-XML stores. This different treatment of "store-to-user" and "user-to-store" information flows fragments service provision into two separate "content presentation" and "content update" infrastructures. Maintaining two disparate service provision implementations that have to evolve in parallel significantly increases the costs for expanding the information space.

Ontologies are widely used to define content conceptualizations in various contexts [43, 44]. Although our approach offers application-specific conceptualizations in a spirit similar to that followed by ontologies, DOLAR: (a) provides a GUI tool for defining conceptualizations in terms of virtual object specifications, (b) generates runtime artifacts that conform to such specifications automatically and (c) enables these artifacts to support two-way data flows atop heterogeneous datastores in a uniform manner. The main difference here is that ontologies use inference as the primary compositional mechanism, while in DOLAR we use DVO instantiation. DVOs allow semantically-diverse and heterogeneous storage artifacts to act as "native objects" of the virtual information space, hiding any underlying physical/storage models. This feature also distinguishes DOLAR from object-oriented databases [45, 46] and various XML object packaging approaches, such as METS [22], MPEG-21 DID [47] and Digital Content Components [48].

Object-Relational Mapping approaches and tools automate the generation of business-logic objects and support bidirectional data flows between such objects and underlying stores [49, 50]. However, such approaches are apparently exclusive to relational databases, while DOLAR can operate atop any heterogeneous datastores that provide a *DOStore* driver. Depending on the operational environment, a particular *DOStore* driver may serve a similar purpose with a mediator [51] or a wrapper [52].

DVO composition schemes offer views/projections of content items and generate such views relying exclusively on runtime objects and not on storage artifacts. We use these schemes as an abstraction tool for separating information utilization options from information access, discovery and conceptualization options, defining the virtual objects' runtime messages/interfaces. To enable application-neutrality and storage-independence, schemes neither get stored in any underlying datastore nor contain any executable code. These characteristics distinguish our scheme-based conception of views from various database or XML view mechanisms that simplify the integration of heterogeneous data [53, 54, 55, 56, 4, 57, 58]. In addition, our scheme-based distinction of application-pertinent and DVO-pertinent compositions draws a significant difference between our composition schemes and disseminator-based approaches [59, 60, 21]. In the latter, storage artifacts are directly associated with repository-pertinent executable code and consequently, there is explicit coupling. The principle of "smart objects and dumb archives" realized through buckets [61] designates a similarity between buckets and DVOs. However, DOLAR storage-independence and scheme-based views are not available in buckets.

Finally, we should also point out that we do not propose the use of DOLAR URIs as global identifiers. In general, supplying content items with global identifiers is considered a best practice, advancing interoperability, especially in domains such as e-publishing. Approaches to offer such identifiers include DOIs [62], Handles [63] and PURLs [64]. Our DOLAR URIs offer a DOLAR-specific means to identify content items, acting as "memory-pointers" in the virtual information space. The use of any particular global identification mechanism is an application-dependent decision that falls outside the scope of DOLAR. In Pergamos, for example, we supply our content items with global HTTP identifiers of the form of **`http://pergamos.lib.uoa.gr/dl/object/uoadl:NUM`** URIs and do not publish any internally employed DOLAR URIs.

Prepared using speauth.cls
DOI: 10.1002/spe

9. CONCLUSIONS AND FUTURE WORK

In protected application environments, information expansion is performed in a controllable fashion, carried out in terms of rigid development procedures. To the contrary, in modern cooperative and public environments such as the Web, information expands more rapidly, constantly "demanding" from applications to catch up with novel types of information. In this article, we presented our DOLAR approach to support information expansion by proposing a virtual information space environment. This environment is based on key automation and abstraction features offered by virtual objects to curtail the costs of introducing new types of items in an existing application. In brief, DOLAR supports uniform two-way data-flows atop any heterogeneous datastores, supporting both accessing and modifying heterogeneous data in an effective way. DOLAR also masks-out the structural diversity of content items using composition schemes, allowing business-logic to operate in isolation of the structural diversity of underlying content. Ultimately, DOLAR achieves separating the information discovery, access, conceptualization and utilization dimensions that compose an information space. This separation enables the independent extension of individual information management options, catering for the inexpensive extension of the information space as a whole. Curtailing the costs of information expansion is valuable for any information-rich application and the operation of DOLAR as the core-mechanism in Pergamos digital library has over time demonstrated its versatility in expanding the information space effectively. As we have shown in this article, the Pergamos business-logic implementation can cope with the addition of new types of content without modifications. In addition, our experiments demonstrated that DOLAR does not impose significant operational overheads even when used atop a variety of heterogeneous data sources.

We plan to extend our DOLAR framework by pursuing work in a number of areas. More specifically, we plan to:

- experiment with our DVO Introspection API. In terms of data definition, this API allowed us to develop an effective GUI tool for issuing virtual object specifications. This tool proved extremely valuable in letting us augment Pergamos with new digital collections in a timely manner. Our plan is to transform virtual object Introspection API into a full-fledged, data-definition DSL [65, 66, 67]. Such a DSL could be embedded in applications to offer a self-contained data-definition facility. This will allow more versatile usages of DOLAR, should we consider the provision of GUI tools tailored to non-technical users such as catalogers and curators. In addition, we plan to implement the embeddable DSL using "just-in-time" compilation of DVO prototypes into Java classes on-the-fly. This will allow for the creation of statically type-safe virtual objects and also offer additional increase in DOLAR performance.
- realize virtual object inheritance. This will empower applications to create new virtual object specifications by building upon existing ones. Supporting inheritance will make virtual objects offer a full object-oriented solution [68].
- continue our work with *DOStore* drivers and introduce a decorator mechanism [13] to allow applications to share/reuse their information access options in diverse operating environments.
- examine data migration capabilities offered by virtual objects. The need of migrating content items to new storage areas is tightly connected to information expansion and occurs in practice frequently. We plan to enrich DOLAR to attain automated migration among heterogeneous stores.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comments that helped us improve the presentation of our work. We also wish to thank G. Pyrounakis, V. Karakoidas and E. Lourdi for their contributions in Pergamos and A. Damianou, V. Nikakis and K. Drakoulakis for their contributions in DOLAR testing and implementation. Finally, we thank A. Avramidis for reading early drafts of this article.

Yannis Smaragdakis was supported by the National Science Foundation under grants CCF-0917774 and CCF-0934631.

DOI: 10.1002/spe

REFERENCES

1. Lesk M. How much information is there in the world? 1997. Technical Report, 1997, Retrieved from `http://www.lesk.com/mlesk/ksg97/ksg.html` on Jan 2010.
2. Lyman P, Varian HR. How much information, 2000 and 2003 2000. Retrieved from `http://www.sims.berkeley.edu/how-much-info` and `http://www.sims.berkeley.edu/how-much-info-2003` on Jan. 2010.
3. Bohn R, Short J. How much information? 2009 report on american consumers 2009. Global Information Industry Center, University of California, San Diego, December 9 2009, Retrieved from `http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf` on Jan. 2010.
4. Lenzerini M. Data integration: a theoretical perspective. *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ACM: New York, NY, USA, 2002; 233–246.
5. Chen K, Chen H, Conway N, Dolan H, Hellerstein JM, Parikh TS. Improving data quality with dynamic forms. *ICTD'09: Proceedings of the 3rd international conference on Information and communication technologies and development*, IEEE Press: Piscataway, NJ, USA, 2009; 487–487.
6. Lukovic I, Mogin P, Pavicevic J, Ristic S. An approach to developing complex database schemas using form types. *Softw., Pract. Exper.* 2007; **37**(15):1621–1656.
7. Parsons D, Rashid A, Telea A, Speck A. An architectural pattern for designing component-based application frameworks. *Softw. Pract. Exper.* 2006; **36**(2):157–190, doi:http://dx.doi.org/10.1002/spe.v36:2.
8. Parnas D. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 1972; **15**(12):1053–1058.
9. Dijkstra EW. Ewd 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective* 1982; :60–66.
10. Tarr P, Ossher H, Harrison W, Sutton S. N degrees of separation: Multi-dimensional separation of concerns. *Proc. of the 21st Int. Conf. on Software Engineering (ICSE)*, 1999; 107–119.
11. Saidis K, Delis A. Type-consistent Digital Objects. *D-Lib Magazine* May/June 2007; **13**(5/6). [doi:10.1045/may2007-saidis].
12. Krasner G, Pope S. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming* 1988; **1**(3):26–49.
13. Gamma E, RHelm, RJohnson, JVlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.
14. Fowler M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
15. Saidis K, Pyrounakis G, Nikolaidou M, Delis A. Digital object prototypes: An effective realization of digital object types. *Proceedings of the 10th European Conference on Digital Libraries*, Alicante, Spain, 2006.
16. UofA. Pergamos Digital Library, University of Athens. `http://pergamos.lib.uoa.gr/`.
17. Saidis K, Delis A. Integrating multi-dimensional information spaces. *2nd Workshop on Very Large Digital Libraries, In conjunction with the 13th European Conference on Digital Libraries*, DELOS Association, 2009.
18. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, 1997; 220–242.
19. DCMI. DCMI Metadata Terms, Dublin Core Metadata Initiative. `http://www.dublincore.org/documents/dcmi-terms/`.
20. Lieberman H. The continuing quest for abstraction. *Proceedings of the 20th European Conference on Object Oriented Programming (ECOOP)*, 2006; 192–197. Doi: 10.1007/11785477_12.
21. Lagoze C, Payette S, Shin E, Wilper C. Fedora: an architecture for complex objects and their relationships. *International Journal on Digital Libraries* 2006; **6**(2):124–138.
22. McDonough JP. METS: standardized encoding for digital library objects. *International Journal on Digital Libraries* 2006; **6**(2):148–158. Doi:10.1007/s00799-005-0132-1.
23. Hamilton, G (Editor). Javabeans specification 1.01. Retrieved from `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html` on Jan 2010.
24. Red Hat Middleware LLC. Hibernate. Available at `http://www.hibernate.org`.
25. DCMI. Simple DC XML schema, version 2002-12-12, Dublin Core Metadata Initiative. `http://dublincore.org/schemas/xmls/simpledc20021212.xsd`.
26. Lagoze C, de Sompel HV. The open archives initiative: Building a low-barrier interoperability framework. *JDCL '01: Proceedings of the 1st Joint Conference on Digital Libraries*, 2001.
27. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W. Getting started with aspectj. *Communications of the ACM* 2001; **44**(10):59–65.
28. Bergmans L, Aksit M. Composing crosscutting concerns using composition filters. *Communications of the ACM* 2001; **44**(10):51–57.
29. Ossher H, Tarr P. Hyper/J: multi-dimensional separation of concerns for java. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2000; 734–737.
30. Ossher H, Tarr P. Using multidimensional separation of concerns to (re) shape evolving software. *Communications of the ACM* 2001; **44**(10):43–50.
31. Sehring HW, Schmidt JW. Beyond Databases: An Asset Language for Conceptual Content Management. *Advances in Databases and Information Systems (ADBIS), 8th East European Conference*, Springer, 2004; 99–112.
32. Arms WY, Blanchi C, Overly EA. An architecture for information in digital libraries. *D-Lib Magazine* February 1997; **3**(2).
33. Suleman H, Fox EA. Designing Protocols in Support of Digital Library Componentization. *ECDL '02: Proceedings of the 6th European Conference on Digital Libraries*, London, UK, 2002; 568–582.
34. Kumar A, Saigal R, Chavez R, Schwertner N. Architecting an extensible digital repository. *JCDL '04: Proceedings of the 4th Joint Conference on Digital Libraries*, 2004; 2–10.

DOI: 10.1002/spe

35. Bainbridge D, Don KJ, Buchanan GR, Witten IH, Jones S, Jones M, Barr MI. Dynamic digital library construction and configuration. *ECDL '04: Proceedings of the 8th European Conference on Digital Libraries*, 2004; 1–13.
36. Kahn R, Wilensky R. A Framework for Distributed Digital Object Services. *International Journal on Digital Libraries* 2006; **6**(2):115–123. Also available at `http://www.cnri.reston.va.us/k-w.html`.
37. Gonçalves M, Fox E, Watson L, Kipp N. Streams, Structures, Spaces, Scenarios, Societies (5s): A Formal Model for Digital Libraries. *ACM Transactions on Information Systems (TOIS)* 2004; **22**(2):270–312.
38. Agosti M, Ferro N. A formal model of annotations of digital content. *ACM Trans. Inf. Syst.* November 2007; **26**(1).
39. CCSDS. Reference Model for an Open Archival Information System (OAIS), Consultative Committee for Space Data Systems 2002. Blue Book, Issue 1, `http://public.ccsds.org/publications/archive/650x0b1.pdf`.
40. Candela L, Castelli D, Pagano P, Thanos C, Ioannidis Y, Koutrika G, Ross S, Schek HJ, Schuldt H. Setting the Foundations of Digital Libraries: The DELOS Manifesto. *D-Lib Magazine* March/April 2007; **13**(3/4). [doi:10.1045/march2007-castelli].
41. Manola F, Miller E, McBride B. Resource Description Framework (RDF) Primer. W3C Recommendation 10 February 2004, `http://www.w3.org/TR/rdf-primer/`.
42. Clark J. XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 Nov. 2009. Retrieved from `http://www.w3.org/TR/xslt` on Jan 2010.
43. Chandrasekaran B, Josephson JR, Benjamins VR. What are ontologies, and why do we need them? *IEEE Intelligent Systems* 1999; **14**(1):20–26.
44. Shadbolt N, Berners-Lee T, Hall W. The semantic web revisited. *IEEE Intelligent Systems* 2006; **21**(3):96–101.
45. Won K. *Introduction to Object-oriented Databases*. MIT Press, 1990. ISBN: 0-262-11124-1.
46. Otis A. A reference model for object data management. *Computer Standards & Interfaces* 1991; **13**(1-3):19–32.
47. Bekaert J, Kooning ED, Walle RVD. Packaging models for the storage and distribution of complex digital objects in archival information systems: A review of MPEG-21 DID principles. *Multimedia Systems* 2005; **10**(4):286–301.
48. Santanchè A, Medeiros CB. A component model and infrastructure for a fluid web. *IEEE Trans. Knowl. Data Eng.* 2007; **19**(2):324–341.
49. Agarwal S. Architecting object applications for high performance with relational databases. *In OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, 1995.
50. Novera JO, Orenstein J, Inc NS. Supporting retrievals and updates in an object/relational mapping system. *IEEE Data Engineering Bulletin* 1999; **22**:22–1.
51. Garcia-Molina H, Papakonstantinou Y, Quass D, Rajaraman A, Sagiv Y, Ullman J, Vassalos V, Widom J. The tsimmis approach to mediation: Data models and languages. *J. Intell. Inf. Syst.* 1997; **8**(2):117–132.
52. Roth MT, Schwarz PM. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1997; 266–275.
53. Gupta A, Jagadish HV, Mumick IS. Data integration using self-maintainable views. *Advances in Database Technology - EDBT '96*, 1996; 140–144.
54. Hull R, Zhou G. A framework for supporting data integration using the materialized and virtual approaches. *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ACM, 1996; 481–492.
55. Ullman J. Information integration using logical views. *Theoretical Computer Science* 2000; **239**(2):189–210.
56. Halevy AY. Answering queries using views: A survey. *The VLDB Journal* 2001; **10**(4):270–294.
57. Vodislav D, Cluet S, Corona G, Sebei I. Views for Simplifying Access to Heterogeneous XML Data. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE. Proceedings, Part I*, Springer, 2006; 72–90.
58. Shao F, Guo L, Botev C, Bhaskar A, Chettiar M, Yang F, Shanmugasundaram J. Efficient keyword search over virtual xml views. *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007; 1057–1068.
59. Blanchi C, Petrone J. Distributed Interoperable Metadata Registry. *D-Lib Magazine* December 2001; **7**(12).
60. de Sompel HV, Bekaert J, Liu X, Balakireva L, Schwander T. aDORe: A Modular, Standards-Based Digital Object Repository. *The Computer Journal* 2005; **48**(5):514–535.
61. Nelson ML, Maly K, Zubair M, Shen SNT. SODA: Smart Objects, Dumb Archives. *ECDL '99: Proceedings of the 3^{rd} European Conference on Digital Libraries*, 1999; 453–464.
62. IDF. The DOI Handbook, The International DOI Foundation. Edition 4.4.1, October 2006, [doi:10.1000/182].
63. CNRI. The Handle System, Corporation of National Research Initiatives. `http://www.handle.net/`.
64. OCLC. Persistent Uniform Resource Locator (PURL), Online Computer Library Center. `http://www.purl.org/`.
65. Hudak P. Building domain-specific embedded languages. *ACM Computing Surveys* 1996; **28**(4es).
66. Wile DS. Supporting the DSL Spectrum. *Journal of Computing and Information Technology* 2001; **CIT 9**(4):263–287.
67. Mernik M, Heering J, Sloane A. When and how to develop domain-specific languages. *ACM Computing Surveys* 2005; **37**(4):316–344.
68. Cardelli L, Wegner P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys* 1985; **17**(4):471–522.

DOI: 10.1002/spe