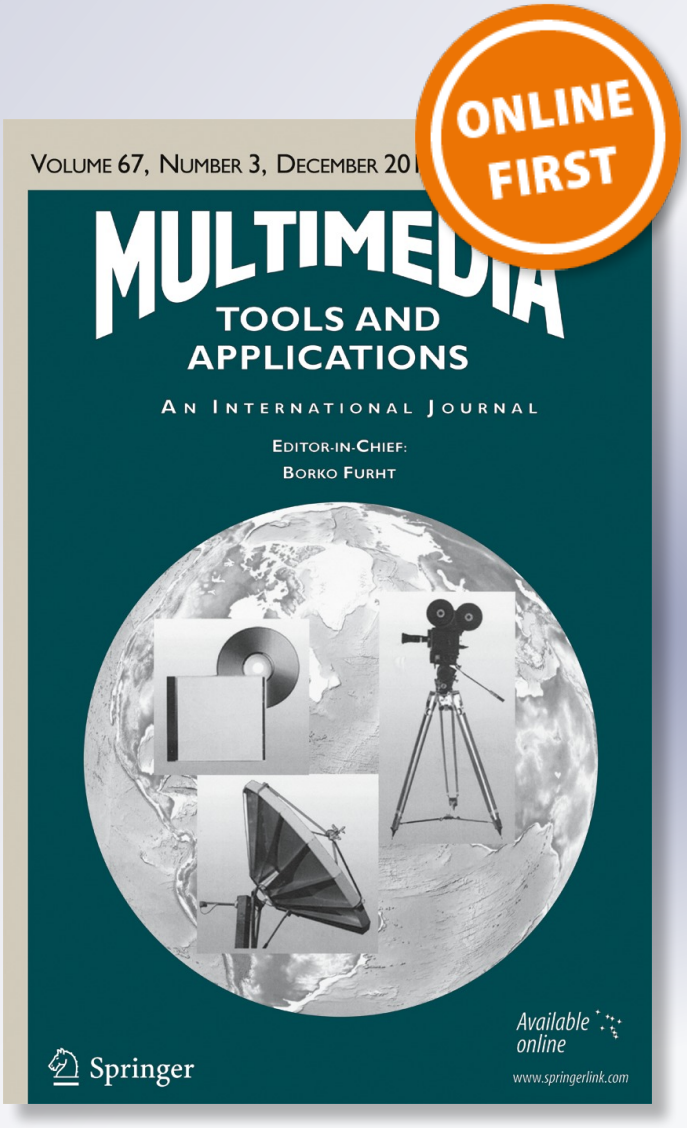
# *EnContRA: a generic multimedia information retrieval meta-framework*

**Ricardo Dias, Manuel J. Fonseca, Nelson Silva & Tiago Cardoso**

**Multimedia Tools and Applications**
An International Journal

ISSN 1380-7501

Multimed Tools Appl
DOI 10.1007/s11042-013-1794-0




Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science +Business Media New York. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Author's personal copy

Multimed Tools Appl
DOI 10.1007/s11042-013-1794-0

# EnContRA: a generic multimedia information retrieval meta-framework

**Ricardo Dias · Manuel J. Fonseca · Nelson Silva · Tiago Cardoso**

© Springer Science+Business Media New York 2013

**Abstract** Over the last years, multimedia collections have largely increased as new items are produced every day, such as pictures, audio/music or video. In Multimedia Information Retrieval, this exponential growth leads content-based approaches to gain advantage over other solutions, not only because they take advantage of the intrinsic information contained in the objects, but also because they automatically process and extract it, reducing the burden taken by developers. Several domain specific frameworks have been developed to efficiently retrieve multimedia items empowering the creation of new content-based applications. However, these frameworks are attached to a specific media type, are too complex to be used in a fast prototyping environment, and are not very flexible nor extensible. To solve these issues, we developed EnContRA, an architectural meta-framework that provides generic building blocks for creating domain specific frameworks. Our meta-framework aims at being ready to be used for fast prototyping, with support for rich and multimodal queries, allowing validation of new descriptors, indexing structures or searching algorithms, while creating domain specific frameworks. In this paper we present the meta-framework architecture and describe in detail its modules and features. To validate the meta-framework, we created an image retrieval framework and a demo application that combines image descriptors with textual information, showing how the hierarchical design of EnContRA could be applied to a searching system and to empower the creation of queries.

R. Dias (✉) · M. J. Fonseca
Department of Information Systems and Computer Science, INESC-ID/IST/ULisboa, Lisbon, Portugal
e-mail: ricardo.dias@ist.utl.pt

M. J. Fonseca
e-mail: mjf@inesc-id.pt

N. Silva · T. Cardoso
inEvo R&D, Lda, Lisboa, Portugal

N. Silva
e-mail: nelson.silva@inevo.pt

T. Cardoso
e-mail: tiago.cardoso@inevo.pt

Published online: 14 December 2013

Author's personal copy



**Keywords** Architectural meta-framework · Multimedia information retrieval · Query processing

## 1 Introduction

Every day people take pictures with their smartphones and digital cameras, record and upload videos to online streaming services and acquire music for only a few cents. These are just a few examples of production and consumption of multimedia items that has taken part of people daily lives over the past years, making the organization, indexing and retrieval tasks harder to do. Therefore, this exponential increase in the size of the multimedia databases created the need for efficient solutions able to deal with these collections.

Content-based solutions have gained much attention from researchers and have been widely explored not only because they take advantage of intrinsic (and extracted) information from multimedia objects, but also because their mechanisms can be performed automatically, making it suitable for large multimedia collections. Several domain-specific and generic frameworks have been created to efficiently retrieve multimedia objects and empower prototyping development. The MESSIF [5] and the OBsearch[1] frameworks are two examples of generic frameworks that allow the development of custom solutions, using indexing and similarity search for different media types. The MARSYAS [17] and Virage [3], on the other hand, are two specific frameworks for audio and image retrieval. Although, these frameworks make the development easier, they still have issues, like for instance, their specificity for a particular media type (e.g., image or audio only) and therefore not adaptable to other types (or to a combination of different types); their difficulty of use, which increases the development effort required to develop new research prototypes and applications; and finally, their flexibility and extensibility on testing and validating specific research components are limited.

To overcome these problems, we propose EnContRA (Engine for Content-based Retrieval Approaches), an architectural meta-framework that provides structure and generic building blocks for creating domain specific frameworks and retrieval applications, with minimum effort. This meta-framework aims at being easy to use for fast prototype research, including validating new descriptors, indexing structures, searching algorithms, etc. Designed as a set of interconnected *empty* containers, defining the main components and their links, EnContRA modular architecture not only offers to developers the common structure and building blocks for typical Information Retrieval (IR) tasks, such as, indexing, searching and querying [4], but also provides the necessary abstraction so they can add new behavior whenever they need. In this paper we describe the meta-framework infrastructure and provide details about modules and features developed to achieve our goals. To validate EnContRA we applied it to a concrete case, by developing an *image retrieval framework* and a demo application, named *Clipart Finder*. In this example we used EnContRA to create a simple query-by-example image retrieval system, exemplifying the usage of the main building blocks, and how we can create modules to fill the EnContRA structure.

In the remainder of the paper, we first give an overview of the related frameworks for content-based retrieval. Section 3 describes the EnContRA meta-framework architecture, its modules and the main building blocks to support the creation of domain specific frameworks. In Section 4 we describe the use of EnContRA for the development of a *image*

[1] http://obsearch.net/

Author's personal copy



*retrieval framework* and a prototype application, *Clipart Finder*. Finally, in Section 5 we conclude the paper and discuss further work directions.

## 2 Related work

Several frameworks, both generic and content-specific, have been proposed to help researchers and developers create content-based solutions. In this section we describe the most relevant works on these two categories, focusing on their scalability, modularity and flexibility.

MUVIS[2] [9] is a generic framework for managing (indexing, exploring, searching, etc.) digital multimedia collections (audio, video and images). This solution provides basic mechanisms to index video in real time and pictures in different formats (JPEG, GIF, BMP, TIFF, etc.). Searching, retrieving and browsing operations are also supported for any of the different types of media. Although, this framework supports typical IR tasks and different media, it is not easy to use nor to quickly create prototypes.

The MUFIN framework (Multi-feature Indexing Network)[3] [14, 15] is another generic multimedia information retrieval framework aiming at being scalable and extensible. This framework is based on the metric space, and therefore being suitable for any metric distance function used in the different scientific research fields, such as biology, geography, etc. To ensure scalability, MUFIN was designed over a Peer-to-Peer (P2P) model, supporting logarithmically upper-bounded routing to peers, in order to process large amounts of data. Performance was also taken into account in MUFIN, by integrating approximate similarity measures and load balancing in the peers. Although, MUFIN provides a generic framework for multiple research fields, it requires a complex infrastructure to hold the framework, making prototyping a hard task.

The MESSIF framework (Metric Similarity Search Implementation Framework)[4] [5], is another framework that explores the metric space to facilitate the development of applications supporting similarity querying. It supports different types of multimedia objects and has a very modular and extensible architecture allowing the easy development of new components, such as, new indexing structures. Moreover, this framework also provides an out-of-the-box set of basic features for storage and automatic performance indicators. A client tool to evaluate and test the indexing structures is also available. Although, conceptually this framework covers the majority of the identified problems, it was not designed to support multimodal queries (using different media types) nor for fast prototyping.

MMRetrieval.net[5] [19] is a multimodal search engine that allows multimedia and multi-language queries, and makes use of the total available information in a multimodal collection. All modalities are indexed and searched separately. Results can be fused with different methods depending on the noise and completeness characteristics of the modalities in a collection, and whether the user is in a need of high initial precision or high recall. Beyond fusion, this engine also provide a two-stage retrieval by first thresholding the results obtained by secondary modalities targeting recall, and then re-ranking them based on the primary modality. The engine demonstrated the feasibility of the proposed architecture and

[2] http://muvis.cs.tut.fi/

[3] http://mufin.fi.muni.cz

[4] http://lsd.fi.muni.cz/trac/messif/

[5] http://www.mmretrieval.net

Author's personal copy



methods on the ImageCLEF 2010 Wikipedia collection. Scalability and performance concerns were also taken into account, by allowing the indexes to be held in different machines. However, the authors were not concerned about providing support for the development of solutions including other types of media nor in extending the current behavior.

Started as a *Google Summer of Code 2007* project, OBSearch is a multimedia search engine written in Java, that aids in creating innovative applications, like for instance, Open Source software license violation, similarity search for music and pictures, etc. This framework is able to deal with computational heavy objects, like graphs and trees, has a very compact API and is very stable and scalable (can be used in a single machine or in network). Though it is a very interesting framework, supporting fast prototyping of solutions, this library was not designed to support the addition of new behavior neither multi-modal querying.

LIRe (Lucene Image Retrieval)[6] [11] is an Open Source library that provides a simple way for retrieving images and photos based on their color and texture characteristics. LIRe uses a Lucene index[7] of image features for content-based image retrieval. Available image descriptors were taken from the MPEG-7 Standard.[8] This library is part of the Caliph & Emir project [10], whose main objective is to easily endow CBIR features in standalone applications. Although, it has a compact API allowing developers to add new descriptors to their specific solutions, their intent is solely to endow CBIR features in innovative applications. Recently, Amato et. al [1], developed a new technique to empower existing *Digital Library Management Systems* with similarity search capabilities, using the Lucene engine, where they combine full-text search with approximate similarity search capabilities, by converting low level features into a textual form.

Virage [3] was a 90's content-based search engine that supported visual queries based on color, composition, texture and structure, by giving different weights to these four features. MARS (Multimedia analysis and retrieval system) [8] was another system for multimedia analysis and retrieval from the 90's, that differed from other systems, as it was an interdisciplinary approach for the integration of computer vision, database management and information retrieval. Its main goal was to organize the various visual features into a meaningful retrieval architecture which could dynamically adapt to different applications and different users. MARS also proposed a relevance feedback architecture in image retrieval while integrating it at various levels during retrieval, including query vector refinement, automatic matching tool selection, and automatic feature adaption.

Regarding music, jAudio [12, 13] is a software library for extracting features from audio files as well as for iteratively developing and sharing new features. These extracted features can then be used in many areas of Music Information Retrieval research, often via processing with machine learning frameworks such as ACE.[9] jAudio current distribution includes 28 implemented basic features and while some of these features are standard features with proven efficacy, others are more innovative and were presented to the research community for experimentation. MARSYAS [17] is an open source audio processing framework with specific emphasis on building Music Information Retrieval Systems. It has been under development since 1998 and has been used for a variety of projects in both academia and industry. The guiding principle behind the design of MARSYAS has always been to

[6] http://www.semanticmetadata.net/lire/

[7] http://lucene.apache.org/

[8] http://mpeg.chiariglione.org/standards/mpeg-7/mpeg-7.htm

[9] http://dtai.cs.kuleuven.be/ACE/doc/

Author's personal copy



provide a flexible, expressive and extensive framework without sacrificing computational efficiency. It has been widely used in Music Information Retrieval to test descriptor extractors and to train classification algorithms. Nonetheless, both frameworks have their focus only on music (audio) and only provide support for very low-level operations.

In summary, *generic-type frameworks* are not suitable for fast prototyping and testing, as their structure and description is complex, and thereby require from developers an extra effort to learn how to use them. However, on the other hand they allow the re-use of their components to develop solutions for different types of media. Media specific solutions are too closely related to the media they handle, and therefore they are not suitable for other media types. So, there is a need for a meta-framework that could easily support the development of domain specific frameworks and the evaluation of new research ideas, with a minimum effort.

## 3 EnContRA

Has stated in the previous section, there has been some effort over the years to develop domain specific frameworks that help developers creating content-based retrieval applications. However, although there is a common structure across these frameworks, little research has been conducted in order to create a generic meta-framework that could easily support the development of domain specific frameworks, and therefore retrieval solutions, with a minimum effort. This is the main driver behind EnContRA.

Developed in Java, EnContRA[10] is an architectural meta-framework that provides generic building blocks which allow the creation of domain specific frameworks and validation of new descriptors, indexing structures or searching algorithms (see Fig. 1). EnContRA architecture can be seen as a set of interconnected abstract containers, defining the main *components* of a framework and their links (*connections*), but not how the containers are filled. This task is done during the creation of domain specific frameworks.

In the next subsections we describe the architecture of EnContRA, its main modules (and building blocks), and finally, how it can be used to create an *image retrieval framework* and a simple demo application (*Clipart Finder*).

### 3.1 Architecture and modules

Figure 1 presents the various levels of EnContRA architecture and shows where the domain specific frameworks and the applications fit, evidencing the EnContRA main role in this process. Our meta-framework, at the left, provides structure and building blocks to developers, so they can create new domain specific frameworks (e.g., an *image retrieval framework*). To perform this task, developers must add specific behavior to the empty building blocks provided by EnContRA (e.g., an image descriptor, like for instance the Scalable Color [10], or a multidimensional indexing structure like the NB-Tree [7]), and describe how they are combined. Finally these specific frameworks can be used to create applications, in a one-to-many schema (see Fig. 1), allowing developers to re-use the artifacts produced. A detailed example of this architecture applied to an *image retrieval framework* is depicted in Fig. 2, and will be explained later in Section 4.

[10] Source and documentation can be found at http://encontra.github.io/

Author's personal copy



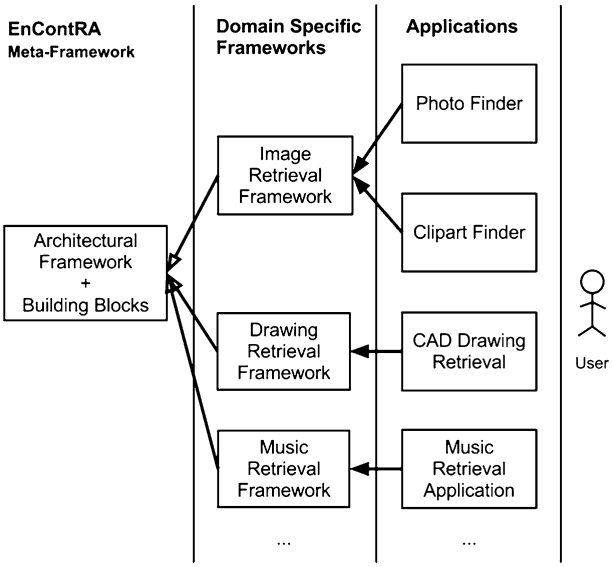



**Fig. 1** Application creation process stack using EnContRA

EnContRA is organized as a set of inter-related modules that separate the different concepts of the meta-framework and that can be combined to create domain specific frameworks. Each module provides a well-defined API to avoid exposing its internal representation and to keep EnContRA extensible. The main modules are organized and distributed around the IR tasks covered in [4], namely, descriptor extraction and transformation from input data; storing and indexing of the extracted information; query processing and searching through the indexes using rich queries. Bellow we briefly describe each EnContRA modules:

- **Storage** - Module that exposes the storage API allowing the use of different storage approaches (for example, a standard database, a memory based storage, etc.).
- **Indexing** - Holds the API for indexing mechanisms and operations, such as, inserting, removing, traversing, etc.
- **Query Processing** - Module that abstracts the query processing mechanism and defines the query operations and operators.
- **Descriptor Extraction** - Module that defines the API for describing descriptors, descriptor extractors, and common methods and operations used during descriptor extraction.
- **Search** - Holds the search API for retrieving information from indexes and storage components.

Notice that these modules only provide structure. Therefore, two aspects must be taken into consideration: to create domain specific frameworks, developers must add behavior to each of the empty building blocks, for example, by defining the descriptors to be used, the indexes, etc; second, performance issues only depend on the components developed for domain specific frameworks that are not part of EnContRA.

Author's personal copy



## 3.2 EnContRA building blocks

The main building blocks included in EnContRA (see Fig. 2) are focused on storing, indexing and searching tasks. In this subsection we provide an overview of each of these modules.

### *3.2.1 Storing*

During the creation of a typical content-based retrieval framework and application, we usually start by defining a data model with the specific properties for the input data collection. In EnContRA this is also the starting point, and a data model can include different media types to describe an object.

Consider for example the context where we are creating an *image retrieval framework*, and we need to define an entity, `ImageModel`, that combines an image and its textual description. To store and retrieve it using a framework developed using EnContRA, we need to implement the *IEntity* interface and annotate the accessor methods with *@Indexed*. By doing this, we are telling EnContRA that these two fields should be used during indexing (more details on Section 3.2.2). The Listing 1 illustrates the definition of this entity.

When passing this entity to EnContRA it knows, from the annotations, that those two fields have to be indexed (image and description), but since EnContRA only provides structure, this definition does not *hard code* the indexes to be used in the framework. This task is left for developers to later add this behavior, which will be described later on this section.

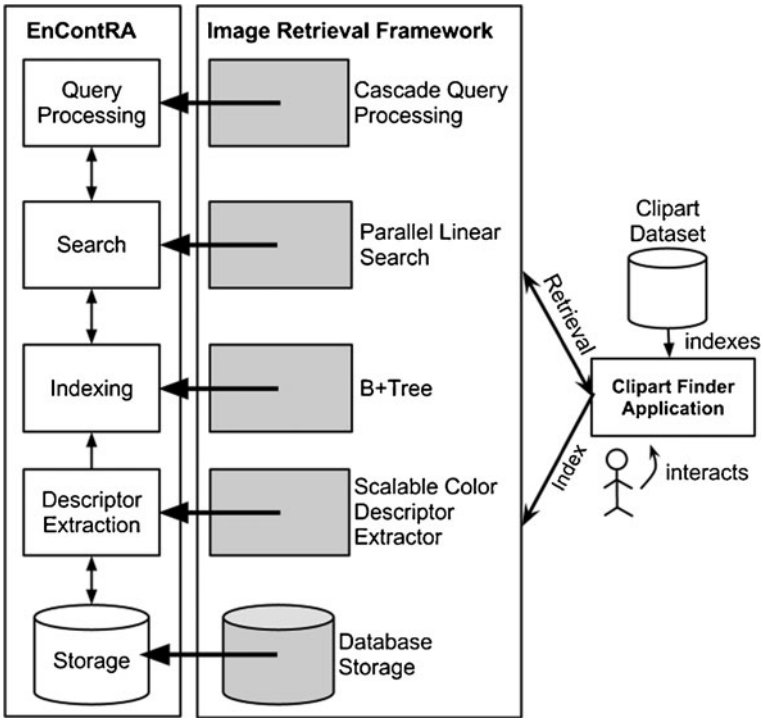



**Fig. 2** Typical architecture of a solution using EnContRA

Author's personal copy



**Listing 1** Example of defining an entity using EnContRA

```
1  public class ImageModel implements IEntity {
2    ...
3    private String description;
4    private BufferedImage image;
5
6    public ImageModel(String description, BufferedImage image) {...}
7
8    @Indexed
9    public String getDescription() {
10     return description;
11   }
12
13   @Indexed
14   public BufferedImage getImage() {
15     return image;
16   }
17    ...
18 }
```

During the indexation of `ImageModel` instances, the EnContRA framework uses a factory to break the entity into smaller objects (called *IndexedObject*). In this example it will be one for the image and another for the description, making it easier to handle by the different indexes. This way, indexes do not deal directly with instances of the data model, but with the actual data they are expecting.

**Listing 2** Example of an EntityStorage definition

```
1  public class MyEntityStorage implements EntityStorage<Long, ImageModel>{
2    ...
3    //stores the instances in memory
4    List<ImageModel> storage = getMemoryStorage();
5    //returns the object given its id
6    public ImageModel get(Long id){
7      return storage.get(id);
8    }
9
10   //saves an ImageModel instance
11   public ImageModel save(ImageModel object) {
12     storage.add(object);
13   }
14
15   //deletes an ImageModel from the storage entity
16   public void delete(ImageModel object) {
17     storage.remove(object.getId());
18   }
19   ...
20 }
```

Author's personal copy



After defining our data model, we must specify how we plan to store it for later use by creating a *EntityStorage*. This entity exposes saving, deleting and retrieving methods, as the example below illustrates:

### *3.2.2 Indexing*

A fundamental task in IR when we want to deal efficiently with large collections is indexing [4]. EnContRA offers an API for defining and creating indexes that exposes common operations they support, such as, inserting and removing entries, iterating, etc. The example below illustrates the Index API:

**Listing 3** Index abstract definition

```
1  public interface Index<E extends IEntry> extends EntityStorage {
2    ...
3    //inserts an element into the index
4    public boolean insert(E entry);
5
6    //removes an element from the index
7    public boolean remove(E entry);
8
9    //gets the first element in the index
10   public E getFirst();
11
12   //gets the last element in the index
13   public E getLast();
14
15   //gets the next element from the index. uses an internal iterator
16   public E getNext();
17
18   //gets the previous element from the index
19   public E getPrevious();
20    ...
21 }
```

Notice that conceptually, indexes are also *EntityStorage* instances, because they share the same operations of the *EntityStorage* API, like the insert, remove and get methods. This means that one might reuse an index implementation for different purposes, such as, a storage mechanism. Also, notice that indexes do not provide direct methods for searching. We took this decision to allow developers to apply different searching strategies regardless of the indexes being used. However, this approach does not prevent developers from extending indexes to embed the searching behavior.

### *3.2.3 Descriptor extraction*

To index data, developers usually start by extracting features from it. These features, called descriptors, describe and give details about a multimedia object. A descriptor is therefore a measurable feature of an object that allows developers to compare objects among them

Author's personal copy



(using similarity measures). EnContRA provides APIs for creating descriptors, extractors, similarity measures, etc. To illustrate this, we present an example of the definition of a new descriptor and its extractor that uses the *euclidean distance* as the similarity measure:

**Listing 4** Definition of a new descriptor and its extractor

```
1  //Example of descriptor
2  public class MyNewDescriptor extends Vector<Double> implements Descriptor {
3    ...
4    protected DistanceMeasure distanceMeasure = new EuclideanDistanceMeasure();
5
6    @Override
7    public double getDistance(Descriptor other) {
8     return distanceMeasure.distance(this, other);
9   }
10
11  @Override
12  public void setValue(Object o) {
13       Vector<Double> val = (Vector<Double>)o;
14       this.setValues(val.getValues());
15   }
16    ...
17  }
18
19  //Example of a descriptor extractor
20  public class MyNewDescriptorExtractor extends DescriptorExtractor<
          IndexedObject<Long, BufferedImage>, MyNewDescriptor> {
21    ...
22    @Override
23    public MyNewDescriptor extract(IndexedObject<Long, BufferedImage> object) {
24
25          MyNewDescriptor descriptor = computeDescriptor(object);
26          ...
27          return descriptor;
28    }
29    ...
30 }
```

Although the previous example is very simple, it highlights some of the flexibility EnContRA allows. More complex and specific descriptors can be implemented in EnContRA, like for example, multiple descriptors for the same object, or composite local descriptors.

Notice that descriptor extraction and similarity measuring are different concepts in EnContRA. This separation allows developers to test and implement different techniques for both extraction and comparison of descriptors. As an example, developers can test different similarity measures for the same descriptors, so they can select the one that gives the best results. Moreover, similarity measures can be more complex than just typical pairwise

Author's personal copy



comparisons, for example, by relying on information extracted from other objects in the dataset, like in the *TF-IDF* algorithm [16].

### *3.2.4 Searching algorithms and query processing*

Querying and searching are other two central tasks of Information Retrieval [4] and a great asset for EnContRA. Stored and indexed data are only meaningful if we can access it easily and efficiently.

Searchers are very tied up to *Indexes* and *Storage* mechanisms, as they are searchable components by nature. Their combination allows developers to create complex frameworks to fulfill their needs. Searching components take a specific query, break it down with the help of *query processors* and lookup for relevant information.

Queries allow us to *retrieve* information from indexes. Creating a query is a fundamental part of how to search through all the data indexed using domain specific frameworks. EnContRA offers a *programmatic query mechanism*, which allows developers to easily create queries without having to learn any kind of Structured Query Language. Queries are created in an object-oriented approach, using a *builder* to specify the criteria to be applied. Below is an example of the creation of a *query builder* and a *query object*, to retrieve *ImageModel* instances:

**Listing 5** Creating a Query Builder

```
1 //creating a builder and a query
2 QueryBuilder cb = new CriteriaBuilderImpl();
3 Query query = cb.createQuery(ImageModel.class);
```

To retrieve data from indexes, developers must add criteria to the queries created, which represents the conditions to be satisfied. To specify the fields of the *data model* to be used as criteria in a query, EnContRA applies a meta-model representation to describe the data model. Taking the `ImageModel` previously described as an example, if we want to create a meta-representation of the *ImageModel* itself, we just have to define a *Path* object (line 2 in Listing 6). To access the image and description fields, we have to obtain the *Path* objects that represent these two fields, from the meta-representation of the *ImageModel* (line 3 and 4 in Listing 6).

**Listing 6** Specifying the fields to be used in a query

```
1 //Path objects are a meta-representation of the entity objects
2 Path<ImageModel> imageModelPath = query.from(ImageModel.class);
3 Path<BufferedImage> imagePath = imageModelPath.get("image");
4 Path<String> descriptionPath = imageModelPath.get("description");
```

In the remainder of this subsection we will complement the querying mechanism by describing the available operators and how to combine them to build complex multimodal queries.

#### Query operators

Query operators allow us to create queries by defining the criteria that constrain the data that will be retrieved. EnContRA offers a query mechanism with support for standard oper-

Author's personal copy



ators (*EQUAL* and *NOT*), predicates (*AND* and *OR*), and a similarity operator (SIMILAR), inspired by the work developed in [1, 2].

While the EQUAL operator is the simplest one and allows the frameworks to look for a specific object or property value in the storage/indexes (just like a Point Query), the SIMILAR operator allows searchers to lookup for objects that are relatively close to the given query, but that are not necessarily the same object (KNN Query) [5, 15].

The *AND* and *OR* predicates allow us to combine expressions, retrieving only the results which logical value respects the expressions and the boolean operator. These predicates accept an arbitrary number of clauses, with a minimum of at least two. Using again the `ImageModel` as an example, imagine we want to look for images containing *red cars* and with the keyword *"me"* in the description. To do this in a domain specific framework created using EnContRA, we have to create two similar expressions (see lines 6 and 7 in Listing 7), one for the picture and another for the textual description, and then combine them using an *AND* predicate (line 10 in Listing 7). Finally, a query is created by filling the *WHERE* clause (line 13 in Listing 7). Figure 3 shows a visual representation of the query described in Listing 7.

**Listing 7** Creating a query with an AND predicate

```
1  //Data
2  String descriptionExample = "me";
3  BufferedImage imageExample = getImageContent();
4
5  //creating the similar expressions
6  Expression descriptionSimilarityClause = cb.similar(descriptionPath,
       descriptionExample);
7  Expression imageSimilarityClause = cb.similar(imagePath, imageExample);
8
9  //combining the similar expressions using AND predicate
10 Expression andExp = cb.and(descriptionSimilarityClause, imageSimilarityClause);
11
12 //create a query with the AND expression
13 Query query = queryBuilder.createQuery().where(andExp);
```

A *NOT* operator is also available and can be applied to any of the previously described operators.

### Query Processor

Queries must be processed so indexes and searching algorithms can make sense of it, and therefore the applications retrieve the information we want to find. The Query Processor is a feature that allows the creation of domain specific frameworks that fit the requirements of their developers, in terms of flexibility and performance. Query Processors define how the queries are broken into smaller pieces that searching algorithms can handle, but also, how the flow of this process unfolds (for example, in a cascade or parallel mode). Query Processors implement the *IQueryProcessor* interface, which the main entry point is the *search* method. Listing 8 shows an example of a cascade Query Processor.

Author's personal copy



**Listing 8** Query Processor Interface

```
1  public class MyQueryProcessor<ImageModel> extends IQueryProcessor {
2    ...
3    //controls the process of breaking the query
4    public ResultSet search(Query query) {
5      //parses the query and obtains a tree representating the query
6      QueryParserNode node = queryParser.parse(query);
7
8      //check if there is an available operator for root tree node
9      if (isOperatorAvailable(node)){
10       //gets the operator and processes the node in a cascade style
11       QueryOperatorProcessor operator = getOperator(node);
12       return operator.process(node);
13     } else {
14       logger.info("No operator was found for: " + node.getName());
15       return new ResultSet<ImageModel>();
16     }
17   }
18   ...
19 }
```

## 4 Test case: image indexing and retrieval

In the previous section we described EnContRA, a generic meta-framework that can enable developers to more easily create domain specific frameworks and applications. We described EnContRA's architecture, its main modules and building blocks that support the creation of solutions that deal with multimodal data.

In this section, we apply EnContRA in a concrete and complete example to evaluate and validate the potential of it. First we develop a framework for *image* retrieval that combines *image* descriptors with textual metadata: then, we describe how this framework can be used to create an example application.

Consider the following scenario: we have a dataset of *cliparts*, and we want to use image descriptors combined with textual metadata to create a framework capable of retrieving cliparts by similarity. This is a typical use of the EnContRA meta-framework, involving

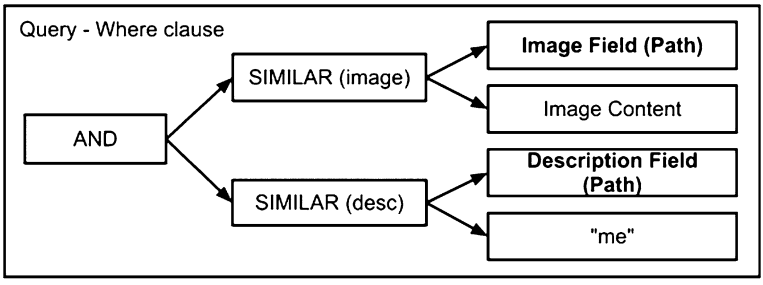



**Fig. 3** Visual representation of a Query composed by an *AND predicate* and two *Similar* expressions

Author's personal copy



querying and retrieving information as the result of combining multimodal queries. The following sections detail how we can use EnContRA to address this problem.

### 4.1 Creating the framework

Based on the scenario defined, we must first develop a framework that will allow us to create the image retrieval application, Clipart Finder. This task consists in filling the building blocks of EnContRA (see Fig. 2) with specific implementations for each one: storage, descriptor extraction, indexing, query processing and searching.

#### *4.1.1 Storage*

Regarding storage, we must first define an *Entity* that describes the data we intent to model. To this end, we can reuse the previously defined *ImageModel* (see Section 3.2.1), that contains two fields, an image (clipart) and a textual description.

**Listing 9** Defining an instance of JPAStorage for ImageModels

```
1  //ID is of type Long and this entity stores ImageModel objects
2  public class ImageModelStorage extends JPAStorage<Long, ImageModel>
       implements EntityStorage<Long, ImageModel> {
3    ...
4    //JPA entity manager
5    EntityManager entityManager;
6
7    public ImageModel get(Long id) {
8     return entityManager.find(ImageModel.class, id);
9    }
10
11   @Override
12   public ImageModel save(ImageModel object) {
13     EntityTransaction tx = entityManager.getTransaction();
14     tx.begin();
15     ImageModel res=entityManager.save(object);
16      tx.commit();
17     return res;
18   }
19
20    @Override
21    public void delete(ImageModel object) {
22         entityManager.remove(object);
23    }
24    ...
25 }
```

Next, we have to define a storage mechanism (*EntityStorage*) for handling *ImageModels*. As previously said, EnContRA only provides containers and links to combine them, so no concrete implementations of this *EntityStorage* is available out-of-the-box. However, for

Author's personal copy



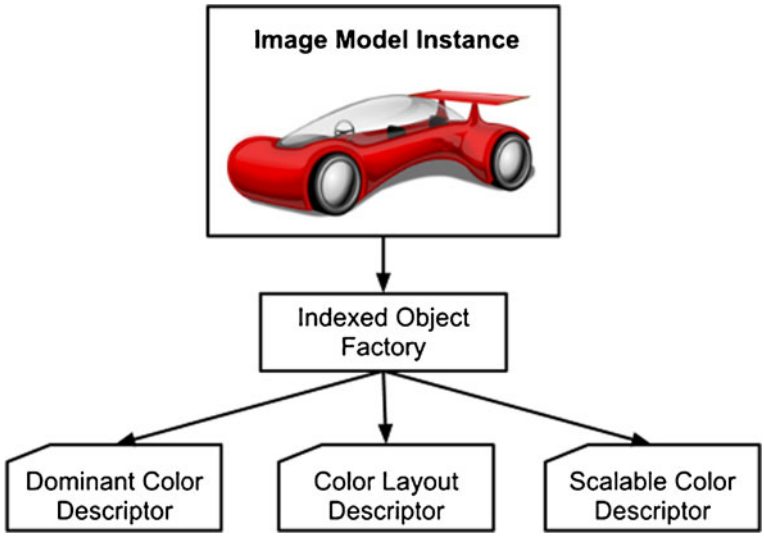



**Fig. 4** Creating descriptors for image

validation and example purposes we implemented two approaches for handling storage: a *memory-based mechanism*, and a *database wrapper*. While the first approach consists in leaving all the information in memory, and it is very useful for rapid prototyping and testing, the second approach consists in storing data into a persistent database, and it is useful when we want to deal with huge amounts of objects. The behavior of this storing mechanism is very similar to the one provided by some frameworks, such as Hibernate[11] or OpenJPA[12], where developers must specify how the data model will be saved by annotating it.

In this example, we choose to use the second approach as it is able to deal with larger data sets (closer to real world constrains). This definition is shown in Listing 9.

### *4.1.2 Descriptor extraction*

Concerning descriptor extraction, we have to focus separately in the different types of data: text and image. As for text, it will be left out, because the chosen implementation of the storage mechanism can effectively handle text indexing and searching. Therefore focus will be on the image feature extraction.

For filling this module in EnContRA structure we implemented a descriptor extraction module for images. In this module we adapted the *MPEG7 descriptors* implemented in LIRe [10], namely, the Scalable Color, the Color Layout and the Dominant Color descriptors. Though we could use only one descriptor for defining this *image retrieval framework*, we decided to combine more than one so when retrieving we could select which ones would be used for comparison between objects (see Section 4.2). Figure 4 shows the three descriptors extracted for this framework.

[11] http://www.hibernate.org/

[12] http://openjpa.apache.org/

Author's personal copy



#### 4.1.3 Indexing

To set up *image indexing*, we have to define the indexing structures to be used and how we want to index the data extracted (the descriptors). In this case, we will assign each descriptor in Fig. 4 to a single indexing structure, so searching can later be performed individually. Here, we used a total of three indexes.

We have implemented three different indexes for validating the EnContRA indexing API: a simple index useful for testing and quick prototyping; a Lucene-based index mechanism to save information [11] (based on the adaptations taken by the LIRe library); and an index using a *B+Tree* [6]. This index uses the B+tree implementation of the *jdbm project*.[13]

For this concrete framework, we decided to use *B+Tree* indexes to index image descriptors (see Listing 10), because it is very efficient while dealing with large datasets.

**Listing 10** Defining a B+Tree Index in EnContra

```
1  public class BPlusTree implements Index<IndexedObject> {
2    ...
3    protected jdbm.btree.BTree btree;
4    protected jdb.helper.TupleBrowser browser;
5
6    public BPlusTree() {
7      btree = new Btree();
8    }
9
10   public boolean insert (IndexedObject entry) {
11     btree.insert (entry.getID(), entry, false);
12   }
13
14   public IndexedObject getNext() {
15       Tuple nextTuple = new Tuple();
16       browser.getNext(nextTuple);
17       return nextTuple;
18   }
19     ...
20 }
```

#### 4.1.4 Query processing

In this test case, query processing shows the capability of EnContRA to deal with multi-modal queries. EnContra Query API allows developers to combine queries for image and text in the same query. Nevertheless, the way queries are broken into searchable pieces for *searching algorithms* is another module developers must fill in.

To develop our domain specific framework, we implemented two approaches for query processing: a cascade version, and a hierarchical / parallel mechanism. While the cascade query processor works like regular cascade algorithms (processing each subquery at a time

[13] http://jdbm.sourceforge.net/

Author's personal copy



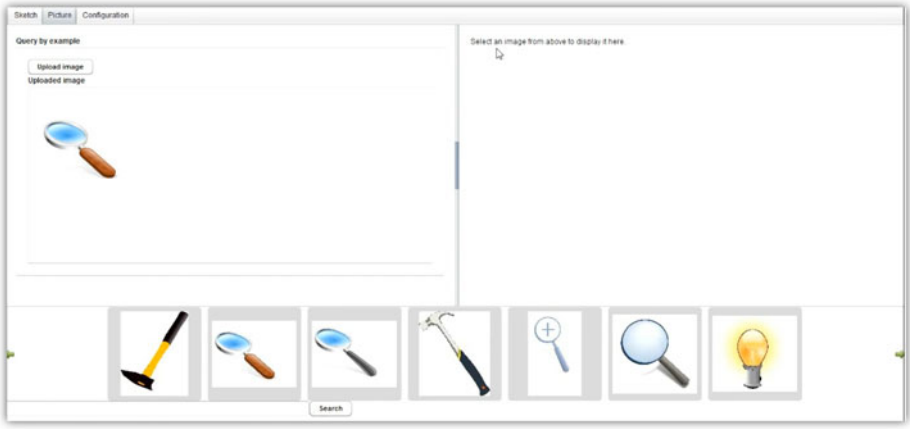



**Fig. 5** Clipart Finder example. Performing a search combining image descriptors and text

in a depth-first style), the second approach works by iteratively processing the query in a hierarchical fashion and searching in parallel whenever it is possible. Taking as an example the query created in the previous subsection (see Fig. 3 as a tree representation of the query), when the processor finds the AND (or OR if available) predicate, it starts processing it in parallel. This is performed at all levels in the query-tree structure, constituting an hierarchical mechanism. Only simple operators, like EQUAL, NOT or SIMILAR are therefore directly handled by concrete searching algorithms.

Regarding result fusion, at the moment it works at every predicate node by applying pairwise combination to the results from each child nodes. However, this mechanism can also take advantage of the hierarchical query processor, because Query Processors at each predicate node could take advantage of results retrieved by other leaves and provide feedback to the remaining ones, to improve performance.

In the current example framework we decided to use the parallelized version of the query processor, whenever it was required, to ensure a good performance of the test case.

### *4.1.5 Search*

Once all the indexing setup is done, including storing and descriptor extraction, we must define how we will perform searching. In this test case, there are two levels of searching: concrete searching algorithms that iterate over the indexes to find relevant data, and structural searching for controlling the searchers at index level.

A possible solution for this two level searching problem is to hierarchically assign one concrete searcher for each index, and a coordinator searcher for dealing with individual image searchers, and working with our *ImageModel* instances (see Listing 11). This solution is scalable and allows parallelization at the different searching levels. Another solution could be to use a single index and searcher, and therefore only one descriptor as the result of combining the three individual descriptors.

As concrete searching algorithms we implemented two approaches: a linear search (parallelized), and an algorithm optimized for multidimensional descriptors using the NB-Tree approach [7]. The *NBTreeSearcher* uses the KNN (k-nearest neighbors) algorithm presented in Fonseca's NB-Tree approach[7] to deal efficiently with high dimensional descriptors. In

Author's personal copy



**Listing 11** Defining how searching will be performed

```
1  //create the top-level searcher for searching
2  Searcher topLevelSearcher = new SimpleParallelSearcher();
3  topLevelSearcher.setQueryProcessor(new QueryProcessorDefaultParallelImpl());
4
5  //create the image searchers for the different descriptors
6  Searcher scalableColorImageSearcher = new NBTreeSearcher();
7  scalableColorImageSearcher.setIndex (new BTreeIndex());
8  //add the searcher to the top-level image searcher
9  topLevelSearcher.setSearcher("scalableColorSearcher", scalableColorImageSearcher);
10 topLevelSearcher.setDescriptorExtractor(new ScalableColorExtractor());
11 ...
12 //do the same for the other image descriptor extractors
```

this implementation, we used a parallel version of the algorithm to improve performance, applying the Actor's Model paradigm.[14,15]

### 4.2 Creating the application

In the previous sections we developed an *image retrieval framework* (see Fig. 2) that can be used for indexing and retrieving similar images, based on queries containing pictures and textual metadata. This framework allows the creation of different applications by loading a concrete dataset, creating a user interface, and so on.

Taking the *image retrieval framework* defined in the previous section we created an application for searching and retrieving cliparts (see Figs. 2 and 5). This application indexes a relatively large clipart database (approximately 26,000 cliparts), and lets users retrieve cliparts from the indexed collection, through the query-by-example paradigm.

Listing 12 shows the creation of a query to retrieve the top-20 most similar images. Figure 5 depicts the resulting application, showing a set of similar results to the submitted query.

**Listing 12** Performing a query-by-example using the *image retrieval framework*

```
1  //creating the query builder, obtain the meta-model representation
2  QueryBuilder queryBuilder = new CriteriaBuilderImpl();
3  Query query = queryBuilder.createQuery();
4  Path<BufferedImage> imagePath = query.from(ImageModel.class).get("image");
5
6  //create the similar expression
7  BufferedImage imageContent = getQueryImage();
8  Expression similarImage = queryBuilder.similar(imagePath, imageContent);
9
10 //retrieve the top-20
11 query = query.where(similarImage).limit(20);
12 ResultSet results = topLevelSearcher.search(query);
```

[14] http://en.wikipedia.org/wiki/Actor_model/

[15] http://akka.io/

Author's personal copy



If we also want to search by the textual description (e.g, a string “me”), we can add a textual field to the previous query, by replacing line 11 by the following one:

**Listing 13** Combining content-based queries with textual metadata

```
1  query = query.where(similarImage).limit(20).storageQuery(“me”);
```

Though in the previous examples we are using all the available descriptors and indexes defined in the framework, developers can still decide and control the ones to be used. This is possible due to the hierarchical structure defined in the framework (see Section 4.1.4). For example, we can create variations of this application by combining the available searchers (and therefore the indexes and descriptors, because of the structure defined in Section 4.1.5), and compare the performance of the solutions developed (see Listing 14).

**Listing 14** Pairwise combination of searchers

```
1  /**
2   *   Available Searchers: scalableColorSearcher, colorLayoutSearcher,
3   *                        dominantColorSearcher
4   */
5  List < Searcher > availableSearchers = topLevelSearcher.getAvailableSearchers();
6
7  //activate two searchers
8  topLevelSearcher.setActiveSearchers(“scalableColorSearcher”, “colorLayout
   Searcher”);
9
10 //perform the query
11 ResultSet results = topLevelSearcher.search(query);
12 print(results);
13 ...
14
15 //activate just one searcher
16 topLevelSearcher.setActiveSearchers(“dominantColorSearcher”);
17
18 //perform the query
19 ResultSet results = topLevelSearcher.search(query);
20 print(results);
21 ...
```

In short, in this section we used EnContRA capabilities to create an image retrieval framework that combines image descriptors with textual information. We showed how the hierarchical design of EnContRA can be applied to a searching structure and how it empowers the creation of queries. Finally, for validation purposes we developed an example application that used this framework for retrieving images from a clipart dataset.

Author's personal copy



## 5 Conclusions and future work

In this paper we described EnContRA, an architectural meta-framework for creating domain specific frameworks. EnContRA offers structure, main building blocks, and connections between them, so developers can create frameworks and applications with minimum effort, by filling EnContRA main building blocks with concrete implementations. It also allows developers to easily test new indexing structures, descriptor extractors and empower the creation of new content-based applications and prototypes for different multimedia object types. Its modular and flexible architecture allows developers to easily extend and add new behavior whenever they intend to. To validate the concepts behind our meta-framework, we developed an *image retrieval framework* and applied it in a demo application for retrieving cliparts, *Clipart Finder*, showing how the hierarchical design of EnContRA can be applied to a searching structure and how it empowers the creation of queries.

As for future work, we are willing to follow three directions: the first one is to work on a specific module for result fusion; the second is to embed a benchmarking module into the meta-framework; and finally, we intent to provide wrappers for other languages more suitable for fast prototyping. Regarding the result fusion mechanism, we intend to take advantage of the result fusion API, by develop and evaluating different hierarchical algorithms not only to speed up result fusion, but also to improve the performance of the whole searching process. We will start by analyzing previous work on this field, such as the one developed by Wimmers et. al [18]. Because performance is essential in Information Retrieval systems and applications, we plan to add a benchmarking tool to help developers analyze the performance of their frameworks and applications and allow direct comparisons in typical tasks, such as, time for indexing activities, time and amount of memory used by a specific searching algorithm, etc. As one of EnContRA main goals is to allow fast prototyping, another work in progress is the creation of wrappers for other languages more suitable for fast prototyping, such as, Groovy, Scala or JRuby. This is possible due to recent efforts to add support for dynamic languages into the *Java Virtual Machine* (JVM).[16]

**Acknowledgments** This work was supported by national funds through FCT –Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013, by ADI through the ColaDI project and through the Crush project, PTDC/EIA-EIA/108077/2008. Ricardo Dias was supported by FCT, grant reference SFRH/BD/70939/2010.

## References

1. Amato G, Bolettieri P, Gennaro C, Rabitti F (2013) Quick and easy implementation of approximate similarity search with lucene. In: Digital libraries and archives, communications in computer and information science, vol. 354. Springer, pp 163–171
2. Amato G, Debole F (2005) A native xml database supporting approximate match search. In: Rauber A, Christodoulakis S, Tjoa A (eds) Research and advanced technology for digital libraries, lecture notes in computer science, vol. 3652. Springer, Berlin, pp 69–80
3. Bach JR, Fuller C, Gupta A, Hampapur A, Horowitz B, Humphrey R, Jain RC, Shu CF (1996) Virage image search engine: an open framework for image management. SPIE, pp 76–87
4. Baeza-Yates RA, Ribeiro-Neto B (1999) Modern information retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston
5. Batko M, Novak D, Zezula P (2007) Messif: metric similarity search implementation framework. In: DELOS07

[16] http://openjdk.java.net/projects/mlvm/

Author's personal copy




6. Comer D (1979) Ubiquitous b-tree. ACM Comput Surv 11(2):121–137
7. Fonseca MJ, Jorge JA (2003) Indexing high-dimensional data for content-based retrieval in large databases. In: DASFAA
8. Huang TS, Electrical MRNO, Engineering C (1996) Multimedia analysis and retrieval system (mars) project. In: Proceeding of 33rd annual clinic on library application of data processing - digital image access and retrieval
9. Kiranyaz S, Gabbouj M (2006) Generic content-based audio indexing and retrieval framework. IEE Proc Vision Image and Signal Process 153(3):285–297
10. Lux M (2009) Caliph & emir: Mpeg-7 photo annotation and retrieval. In: Proceedings of the 17th ACM international conference on multimedia, MM '09. ACM, New York, pp 925–926
11. Lux M, Chatzichristofis SA (2008) Lire: lucene image retrieval: an extensible java cbir library. In: Proceedings of the 16th ACM international conference on multimedia, MM '08. ACM, New York, pp 1085–1088
12. Mcennis D, Mckay C, Depalle P (2005) Jaudio : a feature extraction library. In: International conference on music information retrieval
13. Mcennis D, Mckay C, Fujinaga I (2006) Jaudio: additions and improvements. In: Proceeding of the 7th international conference on music information retrieval (ISMIR), p 385
14. Novak D, Batko M (2009) Metric index: an efficient and scalable solution for similarity search. In: Proceedings of the 2009 2nd international workshop on similarity search and applications, SISAP '09. IEEE Computer Society, Washington, pp 65–73
15. Novak D, Batko M, Zezula P (2009) Generic similarity search engine demonstrated by an image retrieval application. In: Proceedings of the 32nd international ACM SIGIR conference on research and development in information retrieval, SIGIR '09. ACM, New York, pp 840–840
16. Rajaraman A, Ullman JD (2011) Cambridge University Press
17. Tzanetakis G, Cook P (2000) Marsyas: a framework for audio analysis. Organized Sound 4
18. Wimmers E, Haas L, Roth M, Braendli C (1999) Using fagin's algorithm for merging ranked results in multimedia middleware. In: Proceedings. 1999 IFCIS international conference on cooperative information systems, 1999. CoopIS 99, pp 267-278
19. Zagoris K, Arampatzis A, Chatzichristofis SA (2010) www.mmretrieval.net: a multimodal search engine. In: Proceedings of the third international conference on SImilarity search and APplications, SISAP '10. ACM, New York, pp 117–118


**Ricardo Dias** is a PhD student at the Computer Science and Engineering Department at IST/TU Lisbon, Portugal. He is a researcher at INESC-ID Visualization and Intelligent Multimodal Interfaces Group since 2010. He participated in two National projects (ColaDI and Crush). His research interests are in Music Visualization and Retrieval, Playlist Generation and Music Recommendation, and in Indexing Structures that combine multidimensional data and relational databases.

Author's personal copy



**Manuel J. Fonseca** is an Assistant Professor at the Computer Science and Engineering Department at IST/TU Lisbon, Portugal, where he teaches Human-Computer Interaction and Multimedia Information Retrieval. He received a PhD degree in Information Systems and Computer Engineering from IST/TU Lisbon in 2004, discussing "Sketch-based retrieval in large sets of drawings". He has been a member in National and European projects, including SmartSketches, Eurotooling21 and SATIN, being responsible for INESC-ID participation in the last two. From 1998 until now he is a researcher at INESC-ID's Visualization and Intelligent Multimodal Interfaces Group, being responsible for the scientific area of Multimedia Information Retrieval and Visualization. His research interests are in Multimedia Information Retrieval using sketches, User-Centered Retrieval, Interactive Visualization, Human-Computer Interaction and Sketch Recognition. He has published more than 80 technical papers on these and other topics, has participated in more than 20 conference program committees and has been a reviewer for various international journals and conferences. He is Senior member of IEEE, and a member of ACM and EG.

**Nelson Silva** is a Software Engineer and a founding partner of inEvo. He received his BS degree in Information Systems and Computer Science in 2003 from Instituto Superior Técnico, Technical University of Lisbon. He worked as a researcher for the Visualization and Intelligent Multimodal Interfaces Group at INESC-ID from 2002 to 2004 during which he participated in the SmartSketches and Eurotooling21 EC funded projects. He has been in charge of a large number of software projects at inEvo since its beginning in 2004 and his main interests are on web programming, rich interfaces and artificial intelligence.

Author's personal copy



**Tiago Cardoso** is a founding partner at inEvo where he works as a Computer Science Engineer. He received his BS degree in Information Systems and Computer Science in 2003 from Instituto Superior Técnico, Technical University of Lisbon. From 2002 to 2004, he worked as a researcher for the VIMMI Group at INESC-ID having participated in SmartSketches and Eurotooling21 EC funded projects. He has coorientated several MSc thesis and research in areas from rich web interfaces to data visualization. At inEvo, he has developed and participated in several projects and his main interests are on innovative web applications, data visualization and artificial intelligence.