Integration of repositories in Learning Management Systems

José Paulo Leal[1] and Ricardo Queirós[2]

[1] DCC/FCUP & CRACS,
zp@dcc.fc.up.pt

[2] DI/ESEIG & CRACS,
ricardo.queiros@eu.ipp.pt

Abstract. Current Learning Management Systems focus on the management of students, keeping track of their progress across all types of training activities. This type of systems lacks integration with other e-Learning systems. For instance, learning objects stored in a centralized repository are unavailable throughout an organization for potential reuse. In this paper we present a service oriented repository of learning objects called crimsonHex. Its interoperability features are compliant with the existing standards and we propose extensions to the IMS interoperability recommendation, adding new functions, formalizing message interchange and providing also a REST interface. To validate the proposed extensions and its implementation in crimsonHex we developed two repository plugins for Moodle 2.0 that is expected to be included in the next release of this popular learning management system.

Keywords: e-Learning, Repositories, Learning Objects, LMS, Portfolio, Interoperability.

1 Introduction

The main goal of a Learning Management Systems (LMS) is to manage processes regarding the delivery and administration of training and education [1]. Both usage scenarios are relevant: the learners can use the LMS to plan their learning experience and to collaborate with their colleagues; the teachers can deliver educational content and track, analyze and report the learner evolution within an organization. Most LMS's are structured around courses rather than courses' content thus, they only support reusability at the course level. This issue does not allow Moodle users, for instance, to easily bring content into Moodle from external repositories.

This paper builds upon previous work [2] on the design and implementation of crimsonHex - a service oriented repository of learning objects (LO). It provides services to a broad range of e-Learning systems, exposing its functions based on the IMS Digital Repositories Interoperability (DRI) [3] using two alternative web services flavours. Our experience in using these standards lead us to propose

extensions to its set of functions and to the XML binding that currently lacks a formal definition. To evaluate the proposed extensions to the IMS DRI specification and its implementation in the crimsonHex repository, we developed two crimsonHex plugins for the 2.0 release of the popular Moodle LMS. Moodle 2.0 users will be able to download/upload LO from/to crimsonHex repositories, since this LMS is expected to include the plugins described in this paper in its distribution.

The remainder of this paper is organized as follows: Section 2 traces the evolution of e-Learning systems with emphasis on the existing repositories. In the following section we introduce the architecture of crimsonHex and its application interfaces. Then, we provide implementation details of two crimsonHex plugins for Moodle 2.0 using the proposed IMS DRI extensions. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2 e-Learning evolution

e-Learning or Electronic Learning can be defined as the delivery of educational content via any electronic media, including the Internet, satellite broadcast, audio/video tape, interactive TV, CD-Rom and others [4]. Despite some efforts [5] to potentiate remote education, the genesis of e-Learning can be traced with the development of network communication in the late 1960s, more precisely, with the invention of e-mail and computer conferencing (1971). These innovations contribute to the collaboration between teachers and students and initiate a new education paradigm shift [6]. During the 1980s and 1990s, there was a significant growth in the number of students studying part-time and also in non-traditional learners, such as, typical 18/24 years old students seeking the university demand and women's returning to the workforce after child rearing [6]. The growth in lifelong learning has made the educational institutions to seek for flexible education delivery to satisfy these non-traditional students. In the end of the century, this delivery has accentuated with the emergence of new forms of distance delivery based in ICTs advances, such as, the Internet.

In their first generation e-Learning systems were developed for a specific learning domain and had a monolithic architecture. Gradually, these systems evolved and became domain-independent, featuring reusable tools that can be effectively used virtually in any e-Learning course. The systems that reach this level of maturity usually follow a component oriented architecture in order to facilitate tool integration. Different kinds of component based e-Learning systems target specific aspects of e-Learning, such as student or course management. This architectural model structures software around pluggable and interchangeable components, thus enabling the development of larger systems, resulting from the collaboration of different teams. In some cases component oriented architectures led to oversized systems that are difficult to reconvert to changing roles and new demands. This is particularly true in e-Learning. A criticism to this approach [7] is that it reduced e-Learning to the use of one-size-fits-all systems, i.e., systems that 1) can be used on any learning subject but fails to address specific need of each of them, and 2) can be used by any student but is not able to adapt to unique characteristics of individuals.

In parallel with the development component-based systems, practitioners of e-Learning start valuing more the interchange of course content and learners' information, which led to the definition of standards for e-Learning content sharing and interoperability. Standards can be viewed as "documented agreements containing technical specifications or other precise criteria to be used consistently as guidelines to ensure that materials and services are fit for their purpose" [8]. In the e-Learning context, standards are generally developed for the purposes of ensuring interoperability and reusability in systems and of the content and meta-data they manage. In this context, several organizations (IMS, IEEE, ISO/IEC) have develop specifications and standards in the last years [9]. These specifications define, among many others, standards related to learning objects, such as packaging them, describing their content, organizing them in modules and courses and communicating with other e-Learning systems.

The Service Oriented Architecture (SOA) [10] is already a mature architectural pattern with established principles and technologies and can be defined as a systematic approach to system development and integration. The general trend towards SOA was also followed by the e-Learning community. In the last few years there have been initiatives to adapt SOA to e-Learning [11, 12, 13]. These new frameworks and APIs contributed with the identification of service usage models and are generally grouped into logical clusters according to their functionality [14]. For instance, the e-Framework for Education and Research is a joint initiative by JISC, Australia's Department of Education, Science and Training (DEST) and other international partners, to facilitate technical interoperability in education and research fields using SOA.

3 crimsonHex repository

In this section we introduce the crimsonHex repository, its architecture and main components, and we present its application interface (API) used both internally and externally. Internally the API links the main components of the repository. Externally the API exposes the functions of the repository to third party systems. To promote the integration with other e-Learning systems, the API of the repository adheres to the IMS DRI specification. The IMS DRI specifies a set of core functions and an XML binding for these functions. In the definition of API of crimsonHex we needed to create new functions and to extend the XML binding with a Response Specification language. The complete set of functions of the API and the extension to the XML binding are both detailed in this section.

3.1 Architecture

The architecture of the crimsonHex repository can be summarized by the UML component diagram in Figure 1. Using the **API crimsonHex**, the repository exposes a set of functions implemented by a core component that was designed for efficiency and reliability. All other features are relegated to auxiliary components, connected to the central component using this API. Other e-Learning systems can be plugged into

the repository using also this API. Thus, the architecture of crimsonHex repository is divided in three main components:

- **the Core** exposes the main features of the repository, both to external services, such as the LMS and the Evaluation Engine (EE), and to internal components - the Web Manager and the Importer;
- **the Web Manager** allows the searching, previewing, uploading and downloading of LOs and related usage data;
- **the Importer** populates the repository with content from existing legacy repositories, while converting it to LOs.

In the remainder we focus on the Core component, more precisely, its API and we introduce a new language for message interchange.

3.2 Applications Interface

The IMS DRI recommends exposing core functions as SOAP web services. Although not explicitly recommended, other web service interfaces may be used, such as the Representational State Transfer (REST) [15]. We chose to expose the repository functions in these two distinct flavours. SOAP web services are usually action oriented, especially when used in Remote Procedure Call (RPC) mode and implemented by an off-the-shelf SOAP engine such as Axis. REST web services are object (resource) oriented and implemented directly over the HTTP protocol, mostly to put and get resources.

The reason to provide two distinct web service flavours is to encourage the use of the repository by developers with different interoperability requirements. A system requiring a formal an explicit definition of the API in Web Services Description Language (WSDL), to use automated tools to create stubs, will select the SOAP flavour. A lightweight system seeking a small memory footprint at the expense of a less formal definition of the API will select the REST flavour.

The repository functions exposed by the Core component are summarized in Table 1.

Table 1. Core functions of the repository.

Function	SOAP	REST
Reserve	*XML getNextId(URL colectionl)*	*GET URL?nextId > URL*
Submit	XML submit(URL loid, LO lo)	PUT URL < LO
Request	LO retrieve(URL loid)	GET URL > *LO*
Search	XML search(XQuery query)	POST URL < XQUERY > XML GET URL?name1=value1&…> XML
Alert	RSS getUpdates()	GET URL?alert+seconds > RSS
Report	*XML Report(URL loid,Report rp)*	*PUT URL < LOREPORT*
Create	*XML Create(URL collection)*	*PUT URL*
Remove	*XML Remove(URL collection)*	*DELETE URL*
Status	*XML getStatus()*	*GET URL?status > XML*

Each function is associated with the corresponding operations in both SOAP and REST. The lines formatted in italics correspond to the new functions added to the DRI specification, to improve the repository communication with other systems.

To describe the responses generated by the repository we defined a Response Specification as a new XML document type formalized in XML Schema.

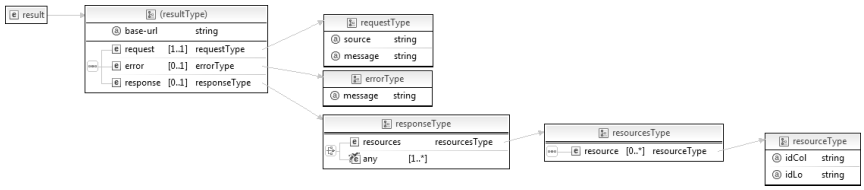

Fig. 1. Response specification schema.

The advantage of this approach is to enable client systems to achieve more information from the server and be able to standardize the parsing and validation of the HTTP responses. Fig. 1 shows the elements of the new language and their types.

The schema defines two top level elements: `result` and `rss`. The former will be used by all the functions except the Alert function that returns a feed compliant with the Really Simple Syndication (RSS) 2.0 specification. The `result` element contains the following child components:

- `base-url` attribute, defining a base URL for the relative URLs in the response;
- `request` element, containing the full request URL and an human readable request message;
- `error` element, containing an error message - client systems will search for this element to verify the existence of errors;
- `response` element, describing a successful execution of the function - it's composed by an human readable response message and, for some functions, by a `resources` element that groups a set of resources defined individually in `resource` elements.

A `resource` element contains an identification of the collection absolute path (attribute `idCol`) and an identification of the LO itself (attribute `idLo`).

In the remainder of this section we enumerate the Core functions of the repository, describing both the request and response data. For sake of simplicity we illustrate the requests using the REST interface since these can be used as command lines in a Linux system shell.

The **Register/Reserve** function requests a unique ID from the repository. We separated this function from Submit/Store in order to allow the inclusion of the ID in the meta-data of the LO itself. This ID is an URL that must be used for submitting or retrieving an LO. The producer may use this URL as an ID with the guarantee of its uniqueness and with the advantage of being a network location from where the LO can be downloaded. This action is performed, for instance, by sending a GET HTTP request to the server, as in the following example.

```
GET http://server/ch/lo?nextId > URL
```

The HTTP response includes an XML file complying with the Response Specification and containing all the details of the response generated by the Core. Nevertheless, in this particular function and for convenience of programmers using REST, the HTTP *Location* header contains the URL returned by the server.

```
Location: http://server/ch/lo/3
```

The **Submit/Store** function uploads an LO to a repository and makes it available for future access. This operation receives as argument an IMS CP compliant file and an URL generated by the Reserve function. This operation validates the LO conformity to the IMS Package Conformance and stores the LO in the internal database. To send the LO to the server we could use, in the REST flavour, the PUT or the POST HTTP methods. An example using the POST syntax is the following.

```
POST http://server/ch/lo/3 < LO
```

The repository responds with submission status data compliant with the Response Specification.

The **Search/Expose** function enables the e-Learning systems to query the repository using the XQuery language, as recommended by the IMS DRI. This approach gives more flexibility to the client systems to perform any queries supported by the repository's data. To write queries in XQuery the programmers of the client systems need to know the repository's database schema. These queries are based on both the LO manifest and its usage reports, and can combine the two document types. The client developer needs also to know that the database is structured in collections. A collection is a kind of a folder containing several resources and sub-folders. From the XQuery point of view the database is a collection of manifest files. For each manifest file there is a nested collection containing the usage reports. As an example of a simple search, suppose you want to find all the titles of LOs in the root collection whose author is Manzoor. The XQuery file would contain the following data.

```
declare namespace imsmd = “http://...”;
for     $p in //imsmd:lom
where   contains($p//imsmd:author,’Manzoor’)
return  $p//imsmd:title//text()
```

After creating the XQuery file you can use the following POST request.

```
POST http://server/ch/lo < XQUERY
```

Alternatively, you can use a GET request with the searched fields and respective values as part of the URL query string, as in the following example.

```
GET http://server/ch/lo?author=Manzoor
```

Queries using the GET method are convenient for simple cases but for complex queries the programmer must resort to the use of XQuery and the POST method. In both approaches the result is a valid XML document such as the following.

```
<result base-url="http://server/ch/lo/">
 <request
  source="http://server/ch/lo/"
  message="Querying repository" />
 <response message="3 LOs found...">
  <resources>
     <resource idCol="" idLo="5">
       Hashmat the Brave Warrior
     </resource>
     <resource idCol="" idLo="123">
        Summation of Four Primes
      </resource>
     <resource idCol="graphs/" idLo="2">
        InCircle
      </resource>
  </resources>
 </response>
</result>
```

The **Report/Store** function associates a usage report to an existing LO. This function is invoked by the LMS to submit a final report, summarizing the use of an LO by a single student. This report includes both general data on the student's attempt to solve the programming exercise (e.g. data, number of evaluations, success) and particular data on the student's characteristics (e.g. gender, age, instructional level). With this data, the LMS will be able to dynamically generate presentation orders based on previous uses of LO, instead of fixed presentation orders. This function is an extension of the IMS DRI.

The **Alert/Expose** function notifies users of changes in the state of the repository using a RSS feed. With this option a user can have up-to-date information through a feed reader. Next, we present an example of a GET HTTP request.

```
GET http://server/ch/lo?alert+seconds > RSS
```

The repository responds with an RSS document.

The **Create** function adds new collections to the repository. To invoke this function in the REST interface the programmer must use the PUT request method of HTTP. The only parameter is the URL of the collection.

```
PUT http://server/ch/lo/newCol
```

The following is an example of the repository response to a create function.

```
<result base-url="http://server/ch/lo/" ...>
  <request
    source="http://server/ch/lo/newCol"
    message="Creating new collection" />
  <response message="Collection created">
   <resource idCol="newCol" idLo=""/>
  </response>
</result>
```

The **Remove** function removes an existent collection or learning object. This function uses the DELETE request method of HTTP. The only parameter is an URL identifying the collection or LO, as in the following example.

```
DELETE http://server/ch/lo/123
```

The following is an example of the repository response to a remove function.

```
<result base-url="http://server/ch/lo/" ...>
  <request
    source="http://server/ch/lo/123"
    message="Deleting a LO" />
  <response message="LO deleted">
   <resource idCol="" idLo="123" />
  </response>
</result>
```

The **Status** function returns a general status of the repository, including versions of the components, their capabilities and statistics. This function uses the GET request method of HTTP, as in the following example.

```
GET http://server/ch/lo?status
```

The repository responds with data compliant with the Response Specification.

4 Integration with Moodle

In this section we validate the interoperability features of the crimsonHex repository by integrating it with Moodle, arguably the most popular LMS nowadays. For this validation we present the new APIs for Moodle 2.0 plugins and we provide implementation details of a plugin for crimsonHex repositories.

The development of this plugin was straightforward. In terms of programming effort we spent half a day to produce approximately 100 new lines of code. This quick and simple integration benefited from the new interoperability features of the repository.

The beta version of Moodle 2.0 is due in February 2010 and will include support for different types of repositories. Several API are already available to enable the development of plugins by third parties to access repositories, including:

- **Repository API** for browsing and retrieving files from external repositories;
- **Portfolio API** for exporting Moodle content to external repositories.

4.1 Repository API

We chose the Repository API for test the integration features of the crimsonHex repository in Moodle. The goal of this particular API is to support the development of plugins to import content from external repositories. The Repository API is organized

in two parts: Administration, for administrators to configure their repositories, and; File picker, for teachers to interact with the available repositories.

To create a plugin for Moodle using the Repository API one must implement a set of related files. The steps are the following:

1. to create a folder for the plugin (*moodle/repository/crimsonHex);*
2. to add to the plugin folder the files
    a. *repository.class.php* – sub-classing a standard API class and overriding its default methods;
    b. *icon.png* – providing the icon displayed in the file picker;
3. to create the language file *repository_crimsonHex.php* and add it to the folder *moodle/repository/lang/en_utf8/.*

The *repository.class.php* is responsible for handling the communication between Moodle and all repository servers of that type. In this case the repository type is crimsonHex but other types are being developed for other types of repository, such as Merlot, YouTube, Flickr and DSpace.

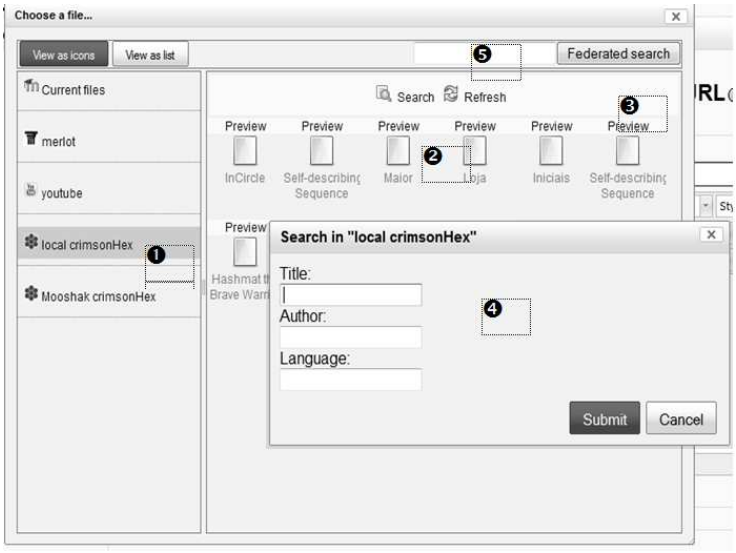

Fig. 2. crimsonHex repository plugin interface.

As explained before, the Repository API has two parts – Administration and File Picker – each with its own graphical user interface (GUI). In Figure 2 we present the file picker GUI of the crimsonHex plugin. On the left panel are listed the available repositories as defined by the administrator. Two crimsonHex repository instances are marked with label 1. Label 2 marks the default listing of the selected repository. Pressing the "Preview" link marked with 3 presents a preview of the respective LO. Pressing the "Search" link pops-up a simple search form, marked as 4 in Figure 3. For federated search in all available crimsonHex repositories is used the text box marked as 5.

For Moodle, each repository is just a hierarchy of nodes. This allows Moodle to construct a standard browse interface. The repository server must provide:

- a URI to download each node (e.g. a LO);
- a list of nodes (e.g. LO and collections) under a given node (e.g. collection).

In addition to these requirements, a repository can optionally support authentication, provide additional metadata for each node (mime type, size, related files, etc.), describe a search facility or even provide copyright and usage rules.

Each feature of the plugin is implemented by a method in the *repository.class.php* file referred in the previous sub-section. A typical method includes: a repository invocation (SOAP or REST), the parsing of its response (using the PHP *simplexml_load_string* function to parse the XML data), a selection of the pertinent data (using XPath) and an iteration over the new results (for instance, populating an array with the relevant data). The next example shows an excerpt of the overridden search function.

```
private function _search($queryString) {
   $list  = array();
   $c = new curl();
   $content=$c->get($this->options['url'] . $queryString);
   $xml = simplexml_load_string($content);
   $result = $xml->xpath("//resource");
   foreach ($result as $entry) {
        $attr = $entry->attributes();
      $list[] = array(
       'title'=>(string)$entry,
       'thumbnail'=>$OUTPUT->icon_url(path),
       'date'=>'',
       'size'=>'',
      'source'=>$attr['url'].$attr['idCol']
      .$attr['idLo']);
   }
   return $list;
}
```

4.2 Portfolio API

The Portfolio API is a core set of interfaces that should be implemented to easily publish files to all kinds of external repository systems. We chose the Portfolio API for test the integration features of crimsonHex repository in Moodle, more precisely, to export Moodle's content to crimosnHex. A typical user story would be:

1. When portfolios are enabled, every page or major piece of content in Moodle would have a "Save" button beside it.
2. User clicks on it and chooses from a list of configured portfolios (this step will be skipped if there's only one).
3. User may be asked to define the format of the captured content (e.g. IMS CP, IMS LD, PDF, HTML, XML).
4. User may be asked to define some metadata to go with the captured content (some will be generated automatically).
5. The content and metadata is COPIED to the external portfolio system.
6. User has an option to "Return to the page you left" or "Visit their portfolio".

To create a plugin for Moodle using the Portfolio API one must implement a set of related files. The steps are the following:

1. to create a folder for the plugin (*moodle/portfolio/type/crimsonHex);*
2. to add to the plugin folder the file *lib.php* – sub-classing a standard API class and overriding its default methods;
3. to create the language file *repository_crimsonHex.php* and add it to the folder *moodle/portfolio/type/crimsonHex/lang/en_utf8/.*

The *lib.php* is responsible for handling the communication between Moodle and all external repository systems of that type. In this case the repository type is crimsonHex but other types are being developed for other types of repository, such as Mahara, GoogleDocs, and Box.net.

This plugin is still in early development and for that reason we only detail the methods that must be overridden in the class `portfolio_plugin_crimsonHex` whose must extend either `portfolio_plugin_push_base` or `portfolio_plugin_pull_base`. The real differences between them are that one pushes the package directly to the remote system (usually through a HTTP POST), and the other type (pull) requires the remote system to request it. The methods are the following:

- `prepare_package`: prepares the package for sending. This might be writing out a metadata manifest file and zip up all the files in a temporary directory. This is called after writing the files out to a temporary location. You can zip files using `$this->get('exporter')->zip_tempfiles`;
- `send_package`: send the package to the remote system. You can retrieve the files the caller has written out using `$this->get('exporter')->get_tempfiles()` which returns an array of stored_file objects;
- `get_interactive_continue_url`: return an URL to present to the user as a 'continue to their portfolio' link;
- `expected_time`: get the transfer time.

5 Conclusions

In this paper we present the interoperability features of crimsonHex - a repository of learning objects. In its current status crimsonHex is available for test and download at the site of the project [16]. The features of crimsonHex were designed based on the IMS Digital Repository Interoperability and we propose several extensions to this specification. These extensions include new functions and a formal definition of a response specification for the complete function set. To evaluate these extensions we implemented two plugins for the 2.0 Moodle's release that uses the interoperability features of crimsonHex and will facilitate the use of crimsonHex by Moodle users.

The main contributions of this work are the proposed extensions to the IMS DRI specification and the two plugins to be included in the Moodle 2.0 distribution.

Completing the implementation of the Portfolio API is the next step in this research. This Moodle API is in early development cycle and we are looking forward to finish the plugin.

Adding authoring features to the crimsonHex is other research path. Creating LOs with metadata of good quality is a challenge since the typical author of e-Learning

content usually lacks the knowledge of metadata standards. This is also an interoperability issue since the LMS is where e-Learning content is tested and used in first place but repositories are the appropriate place to promote content reuse as LOs.

Using the plugin based on the Portfolio API will enable the content author to upload learning content to crimsonHex and create a new LO with the essential metadata. Them, using the authoring features of crimsonHex, the content author will be assisted in refining the LO metadata.

References

1. Harman, K., Koohang, A., Learning Objects: Standards, Metadata, Repositories, and LCMS, Informing Science Institute, Edição de Informing Science, 2007. ISBN 8392233751, 9788392233756.
2. Leal, J.P., Queirós, R., 2009. CrimsonHex: a Service Oriented Repository of Specialised Learning Objects. In: *ICEIS 2009: 11th International Conference on Enterprise Information Systems, Milan*.
3. IMS DRI - IMS Digital Repositories Interoperability, 2003. Core Functions Information Model, URL: http://www.imsglobal.org/digitalrepositories.
4. Tastle, J., White, A. And Shackleton, P., E-Learnintens g in Higher Education: the challenge, effort, and return of investment, International Journal on ELearning, 2005.
5. Harasim, L.: History of E-learning: Shift Happened, The International Handbook of Virtual Learning Environments, Springer, 2006.
6. Williams, J., Goldberg, M.: The evolution of e-learning. Universitas 21 Global, 2005.
7. Dagger,D., O'Connor, A., Lawless,S., Walsh,E., Wade,V.: "Service-Oriented E-Learning Platforms: From Monolithic Systems to Flexible Services," IEEE Internet Computing, vol. 11, no. 3, pp. 28-35, May/June 2007.
8. Bryden, A. Open and Global Standards for Achieving an Inclusive Information Society.
9. Friesen, N. Interoperability and Learning Objects: An Overview of E-Learning Standardization". Interdisciplinary Journal of Knowledge and Learning Objects. 2005
10. Earl, T.: Service-oriented architecture - Concepts, Technology and Design, Prentice Hall, 2005, ISBN 0-13-185858-0
11. C. Smythe, “IMS Abstract Framework - A review", IMS Global Learning Consortium, Inc. 2003.
12. Open Knowledge Initiative Website, http://www.okiproject.org.
13. S. Wilson, K. Blinco, D. Rehak, “An e-Learning Framework” - Paper prepared on behalf of DEST (Australia), JISC-CETIS (UK), and Industry Canada, July 2004.
14. S. Aguirre, J. Salvachúa, A. Fumero, A. Tapiador. "Joint Degrees in E-Learning Systems: A Web Services Approach". Collaborative Computing: Networking, Applications and Worksharing, 2006.
15. Fielding, R., 2000. Architectural Styles and the Design of Network-based Software Architectures, Phd. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation.
16. crimsonHex – Project Web site, 2009. URL: http://www.dcc.fc.up.pt/crimsonHex.