

Engineering a Replay Application Based on RDF and OWL

Chris Greenhalgh, Andy French, Jan Humble, Paul Tennant

School of Computer Science and IT, University of Nottingham

email: {cmg, apf, jch, pxt}@cs.nott.ac.uk

Abstract

Digital Replay System (DRS) is an application for replaying and analyzing combinations of video recordings, transcriptions and system log files. The W3C's Resource Description Framework (RDF) and Web Ontology Language (OWL) are used to define and maintain the application's internal data model, based on the JENA RDF library. Each application has its own persistent RDF models, provided by a database-backed JENA model. A top-level division is made into "projects", each of which has its own RDF model for scalability and manageability. An in-memory model cache and carefully limited use of inference yields acceptable interactive performance. OWL is used for data modeling, both for primary data objects used in the application, supported by a Java wrapper generating tool, and for more general descriptive metadata, accessed via ontology-driven generic interfaces. The RDF data model is complemented by relational databases for storing system events and XML-encoded files for editable "rich" documents. This has proved to be an effective – and transferable – implementation and data storage approach. DRS is available under an open source license.

1. Introduction

Digital Replay System (DRS) is an application for replaying and analyzing combinations of video recordings, transcriptions and system log files. It is being developed within the Digital Records for e-Social Science (DReSS) node of the National Centre for e-Social Science (NCeSS). It is currently being used to study and analyse: ubiquitous computing systems in use, learning through simulations, and everyday conversational English including gestures.

DRS builds on previous work in the VidGrid ESRC e-Science pilot project. One of the fundamental changes between VidGrid's Replaytool and DRS has been to completely re-engineer the application to use the W3C's Resource Description Framework (RDF) and Web Ontology Language (OWL) to maintain the application's internal data.

This paper describes some of the issues encountered and the strategies that have been effective in engineering the final application. These issues and strategies are relevant to general application development based on an RDF/OWL information model.

Section 2 briefly describes relevant elements of Replaytool and DRS. Section 3 introduces

the use of RDF within DRS. Section 4 describes various performance-related optimizations in the DRS's use of RDF. Section 5 describes how OWL is used. Section 6 overviews DRS's complementary use of relational databases and files. Section 7 identifies areas of future work and the availability of DRS.

2. Replaytool and DRS

Replaytool [1] was a heavy-weight desktop application written in Java. It supported the synchronisation, replay and analysis of a set of related video, audio, text and system log files. Each such set of files was represented within the application by a FileSetModel data object. This was persisted to a simple XML file schema (supported by auto-generated Java API for XML Binding – JAXB – helper classes). This data object maintained minimal metadata about each file in the replay session, including any timing offset and its MIME type.

Replaytool also defined a time-based file viewer Service Provider Interface (SPI), which supported synchronised playback of multiple files of different types. Replaytool included viewers for video files, audio files, time-stamped text files (derived from system or application logs, or human transcription or

annotation) and binary system logs from the EQUIP platform.

Replaytool has been developed and extended in various ways. Within VidGrid it was extended to support distributed synchronous collaborative use for distributed data sessions. The MultiMedia Grid (MiMeG) node of the NCeSS has extended this to support further distributed collaboration and also multi-modal interaction [2].

Within the DReSS node we are interested in supporting more of the recording and analysis process, and a broader range of analytical methods and approaches. Consequently we have needed a much broader and more open information model within the application, e.g. preserving provenance information about sources of video recordings, and various kinds of annotations and codings of observed actions. To facilitate this we decided to migrate the application's core data model to RDF [3] and OWL [4]. We chose an RDF-based approach (rather than an object, XML or Relational model) as it allows any item within the data model to have further information associated with it, and makes it relatively easy to use multiple schemas (ontologies) together within the same application.

Figure 1 shows a typical view of the DRS desktop application, including graphs of logged sensor data (top left), two video records (centre left), playback controls (bottom left), time-line overview (centre right) and transcription (bottom right) (this data is from the “Thrill” project, which is exploring the nature of thrill using theme park rides).

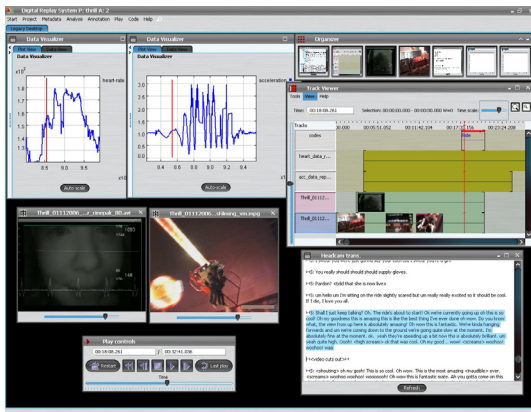


Figure 1. Typical DRS session.

3. Using RDF and OWL within DRS

Having used – and been happy with – HP Lab’s open source JENA [5,6] RDF library in

previous projects we adopted this as the core RDF store and API for DRS.

Because some of our users require the ability to work while on the move and not necessarily networked (e.g. on a train) each DRS desktop application must have its own RDF store, local to that particular machine.

3.1. Models

We decided from the outset to organise the information in DRS as multiple independent RDF models. This is reflected in DRS by the provision of top-level “Projects”, which are nominally independent units of potentially collaborative work. Each project’s data is stored in a single JENA RDF model. This separation reduces the scalability problems to be addressed (if a project gets too large and slow then some of the work can be migrated to a new project), and provides a top level division for management and security (e.g. only certain users may have access to certain projects).

In addition to the project models, each DRS application also has a single “index” model which it uses to store project-independent configuration information and to provide an index to the project-specific models.

3.2. URIs

In RDF each *resource* (each data ‘object’) is identified by a Uniform Resource Identifier (URI). DRS uses URIs which resembles HTTP URLs. In standalone use the administrator installing the application must provide a unique URI prefix for that application. In networked use the workgroup server allocates unique URI prefixes to each DRS client. This prefix and the next URI to be allocated are maintained persistently in the index RDF model, so that new URIs can be allocated by the DRS application without further network communication.

3.3. Anonymous Nodes

RDF allows unnamed resources, i.e. resources or “things” with no URI. However, in DRS every resource must have a distinct allocated URI and anonymous resources are not used. This is to avoid the problem that otherwise occurs when transferring such resources to another application (such as a DRS server or another client) and back again: it is no longer possible to tell which anonymous node is which, and in particular whether it is the “same” anonymous node or not. Naming all resources makes the model a little less compact but avoids

this ambiguity, making exchange and editing of RDF fragments more straight-forward.

4. Performance

4.1. Model Cacheing

JENA supports a number of different RDF Model implementations, ranging from transient in-memory models to persistent models backed by a relational database. Our first approach was to use the provided database-backed persistent models with a MySQL database. However we found that the update performance was extremely slow (about 30ms per RDF statement added to the persistent model in initial tests).

The first performance enhancement we made was to use an in-memory cache for the project model and only to persist changes when explicitly requested by the user (i.e. on save or exit). This approach makes the persistence delays more anticipatable and therefore more manageable (for the user). It also provides a possible building block for a simple undo/redo facility. However, it also leaves recent changes non-persistent and so vulnerable to loss, e.g. if the application crashes or is terminated unexpectedly.

The in-memory cache makes use of JENA's provided MonitorModel implementation (package com.hp.hpl.jena.util); this class sits on top of another model implementation and allows the lists of statements being added and removed to be polled. The implementation structure of this caching strategy is shown in figure 2.

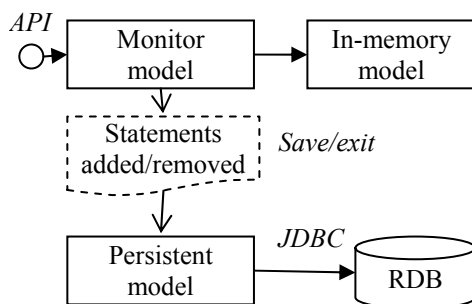


Figure 2. In-memory cacheing of persistent RDF model

This approach is logically related to the insert batching described in [8], but is applied here to interactive use of a graphical application rather than to bulk updates of a central repository.

4.2. Choice of RDBMS

The time taken to persist changes was still found to be unacceptably long during interactive

sessions (e.g. tens of seconds or minutes when importing a new transcript). So we also tried changing the RDBMS from MySQL to the Hypersonic SQL Database (HSQLDB) [7] (at the recommendation of the Replayer ESRC small grants project). This is a Java database, with file-based persistence, which can be used as a server or within a single Java application. The latter was ideal for use within DRS, since we expect only a single DRS application to be running at any time on the desktop machine. This also considerably simplifies the installation process: MySQL required a substantial separate installation and configuration process prior to the use of DRS, whereas simple file-backed use of HSQLDB just requires access to a writeable folder in the filesystem. However, HSQLDB must be explicitly shut down by DRS as the application closes or the last few changes are lost (at least with the default configuration).

In initial tests we found performance with HSQLDB to be much better than with MySQL. For example, persisting a newly imported transcript (with 88 timed speech segments, in total encoded as 2130 RDF statements) took around 54 seconds (25ms per statement) in MySQL but only 0.48 seconds (0.22ms per statement) in HSQLDB. However, we also found that HSQLDB performance fell approximately linearly as the model size increased, for example the same action took 2.6 seconds or 1.22ms per statement with about 13,000 statements in the model (see also the next section for qualification of this). Performance with MySQL was approximately constant up to this model size (and others have used JENA/MySQL with multi-million statement models, e.g. [8]).

4.3. Duplicate Checking

The performance achieved with MySQL (above) for model update is much slower than that reported in the JENA documentation: [9] reports approximately 1.1ms per statement for adding to a JENA/MySQL persistent model for 1000+ statements. The JENA documentation suggests that the default duplicate statement checking can be disabled in order to increase model building time, provided that there are no duplicate statements being added to the model, although their indicative results [10] suggest approximately a factor of 2 effect on performance.

When we tried disabling duplicate checking in our application we found that insert performance with MySQL was approximately 0.75ms per statement and with HSQLDB 0.33ms per statement for the transcription

import considered above. In both cases the performance does not degrade noticeably as the model size increases (at least up to ~15,000 statements). In other words, duplicate checking speeded up inserting statements to the MySQL-backed model by 97%, and removed the strong model-size effect observed with HSQLDB. We are uncertain why duplicate checking has had such a large impact on our application compared to [10]. Possible factors include: changes in JENA (we are using JENA 2.4 at present); higher incidence of object-valued properties; use of relatively long resource URIs.

Note that duplicate checking should only be disabled if it is known that duplicates are not present in the statements being added (which is not the case in general). However, with the model caching approach described in section 4.1 the monitor model returns minimal lists of statements added and removed which therefore will never duplicate statements already in the persistent model. Consequently duplicate checking is not required with the model cache approach.

At present we are retaining HSQLDB as the default database solely because of the simpler installation requirements.

4.4. Inference

JENA supports various levels of inference within RDF models, ranging from none, though simple RDFS entailments [11] to a substantial subset of OWL-Lite. Custom inference rules can also be specified. For example, a typical entailment (or inference rule) is that if class A is a subclass of class B and B is a subclass of class C then A is also a subclass of C.

We anticipate that the RDF models in DRS may grow to tens or hundreds of thousands of statements, and consequently we were very cautious about requiring inference at run-time due to performance concerns. Our collaborators in the myGrid project also experienced particular performance problems attempting to combine inference with persistent models in JENA.

Consequently we have designed the system to require almost no inference. The only inference that is performed is on the model which holds the system's ontology (only) in memory. This is limited to simple RDFS entailments, in particular, subclass relationships as in the example above. Even attempting to use the OWL-Lite entailments on this model resulted in simple queries taking around a second (the ontology currently has around 220 classes).

This approach requires us to separate the ontology model from the base data model (as described previously). This makes some queries and operations more awkward than combining the two, but maintains performance and allows us to more easily evolve the ontology separately from the base data. For example, determining all individuals belonging to a super-class requires first that all possible subclasses be determined from the ontology model (which has simple entailments such as this), and then the base model can be queried to identify individuals of these classes.

5. Using OWL

The ontology for DRS has been specified using the Web Ontology Language (OWL), in its intermediate complexity variant OWL-DL (DL=Description Logic). However relatively few OWL features are actually used. The Protégé tool [12] has been used to author the ontology.

5.1. Organisation

The ontology is currently divided into four sub-ontologies:

- digitalrecord.owl - core digital record,
- replaytool2.owl - DRS configuration and security,
- guiconfiguration.owl - new GUI configuration options, and
- logfileworkbench.owl - for working with system log files and databases.

One reason for this separation is simply to have a number of separate ontology files which can be edited independently. Making even small changes in Protégé can result in large changes in the resulting OWL files, due to slightly different orders of traversing the ontology model during file writing. These large changes make it relatively problematic to merge changes between concurrent edits in a source-code control system such as CVS (which we use for developing DRS).

5.2. Ontology “Wrapper” Classes

Some of the classes defined in the ontology are solely or largely descriptive metadata, e.g. about people involved in studies, or data capture sessions. Other ontology classes reflect core elements of the functionality of the DRS application, e.g. access rights, media files, projects and analyses (file sets).

For the latter ontology classes it is very difficult to use the standard generic JENA RDF model API within DRS. The generic API has no

tailoring to any particular ontology, and operates simply at the level of RDF resources, properties and statements (the combination of a resource, a property and a value). Using this generic API results in relatively large amounts of code to perform relatively simple operations on the information in the model (e.g. to get or set a project access time), gives weakly typed results and is therefore more prone to bugs and run-time errors, and does not enforce ontology constraints during programming (e.g. subclass relationships, property domains and ranges).

Instead we make use of tool-generated Java “wrapper” classes to represent ontology classes within these parts of the DRS application. At the time we looked for existing tools, and evaluated Kazuki [13] in particular. However we were unable to find a tool that supported up-to-date versions of JENA and reflected the ontology class hierarchy in the generated Java class/interface hierarchy. Consequently we wrote our own wrapper generator, “wrappergen”. This supports a useful subset of OWL:

- Each ontology class generates a correspondingly named Java interface (it must be an interface in order to support multiple inheritance).
- This interface defines getter and setter methods for each property specific to that class, as determined by the property domains. These methods are inspired by those generated by JAXB, allowing a property to be unset as well as set.
- Java 1.5 generics are used to strongly type set-valued properties. Which properties are not set-valued is determined from functional properties and from cardinality restrictions.
- Property value Java types are determined from the property ranges, with both datatype and object ranges supported (object ranges are mapped to the generated Java interfaces where possible).
- Each ontology class also generates a corresponding implementation class, which implements the corresponding Java interface and all super-interfaces. The implemented property getter/setter methods in turn make use of the generic JENA model API to actually get/set values and resource references.

Each Java object corresponding to an RDF individual is a subclass of a Thing class, which keeps track of the individual’s URI and the JENA model which it is in.

A supporting class for each ontology also establishes a mapping between ontology class

URIs and Java classes, allowing resources from the RDF model to be dynamically wrapped to the most appropriate (i.e. most derived) Java class.

5.3. Example

For example, in DRS each video file is represented in the RDF model by an individual of ontology class `dr:Video`. This is a subclass of ontology class `dr:Media`. Various properties are defined for `dr:Media`, including a multi-slot string-valued datatype property `dr:hasMimeType`.

Figure 3 shows the built-in `Thing` and `ThingImpl` top-level interface and class which are part of the wrappergen support library, and the generated `Media` and `Video` interfaces and corresponding `MediaImpl` and `VideoImpl` implementation classes. Note that there is no inheritance relationship between implementation classes because Java will not allow multiple inheritance between classes (only between abstract interfaces).

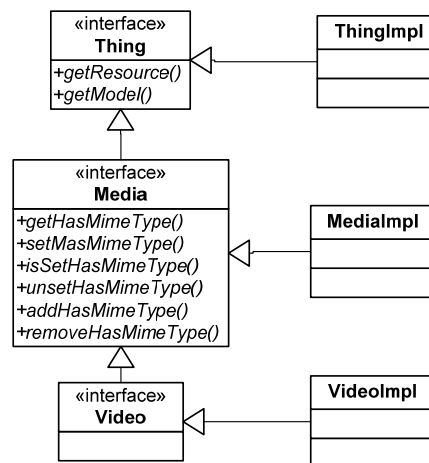


Figure 3. Example generated wrappers

We have found this approach produces more readable code and allows more checking to be performed by the compiler. For example, substantial changes to the ontology would result in compilation failures rather than runtime failures.

The generated wrapper classes currently have no support for generating more complex queries, so the JENA Query API (which supports the SPARQL RDF query language) must be used in these situations. The resulting resources can then be wrapped in the appropriate Java objects for further custom processing.

5.4. Generic Graphical Interfaces

The Java wrapper classes are very useful for coding application logic against specific ontology-defined classes, but are less useful when working with purely descriptive metadata which has no custom logic. To support such metadata DRS includes a number of general-purpose graphical interfaces which use the generic JENA API together with run-time inspection of the ontology. These include:

- A properties frame, which shows the RDF properties of the current selection (if any);
- An instance list frame, which shows instances of an ontology class (including subclasses);
- A new instance dialog, which allows a specific ontology class to be selected from a portion of the ontology to create a new individual;
- A value table frame, which gives a tabular view of all individuals of a particular class and their property values, or of all values of a particular object-typed property (see figure 4);
- A metadata explorer, which shows an expandable tree-view rooted in a single individual and showing its object-valued properties, and so on.
- A datatype property editor frame, which allows datatype-valued properties of an individual to be viewed and edited in a datatype-specific manner (e.g. constraining integers to be integers).

Instance	hasFilename	hasMimeType	title	version
doc1 : DRDocumentLiteral (...)			doc1	
activity16.gif : Image (http://ku...)	activity16.gif	image/gif	activi...	
licence.txt : Text (http://ku...)	licence.txt	text/plain	licenc...	
Demo.mpg : VideoCapture (...)	Demo.mpg	video/mpeg	Demo...	

Figure 4. Generic individual table view.

These interfaces allow free exploration and editing of a project’s RDF model. However, they require a substantial understanding of the nature of RDF and the particular area of the ontology that is of interest. To support more general use we are exploring more tailored and specialised versions of these interfaces, sacrificing some generality for greater usability. For example, a tailored view of projects and analyses is given by the Project Browser (figure 5) which is a key visualisation of current DRS use.

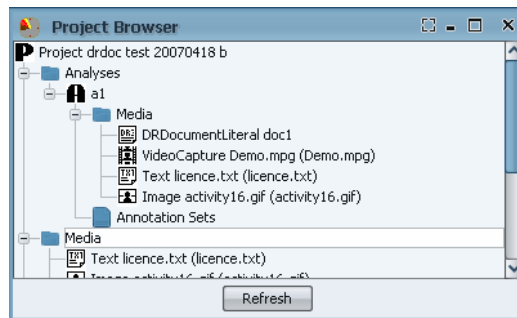


Figure 5. Project browser view.

6. Integrating Files and Databases

In DRS we have sought to use RDF wherever possible for data storage and manipulation. For example, the internal project index, access controls and GUI configuration are all modelled and persisted in the RDF model(s). However, RDF has not provided a complete solution for data storage.

From the outset we have had large media and log files to work with in DRS, e.g. multiple hours of digital video from multiple cameras. It is not currently feasible – or appropriate – to convert these to an RDF representation for normal use. In the case of media files, these are managed as files and the Replaytool viewer framework plays directly from these files; the RDF model contains suitable metadata to describe and support this.

In addition we have found two further situations in which we have adopted alternative and complementary storage technologies: relational databases for system logs; and XML documents for editable documents.

6.1. Relational Databases for System Logs

A typical system or application log file may contain hundreds, thousands or even millions of relatively uniform system event records, e.g. recording the value of a particular sensor, the activation of a particular program option or the current value of an internal variable. It is possible in principle to transform such events and logs into RDF. However the size of the resulting model would be large compared to the current metadata (e.g. millions of statements for many of our current log files), and the interactive performance is likely to be correspondingly poor.

In the first phase of prototyping in the DReSS node we extended the original Replaytool application with a “Data Goggles” application (see [1]) which managed and visualised a database of processed system events in a set of relational tables. In a similar

way, the Replayer tool [14], now being developed in an ESRC small grant project at Glasgow, is built around a relational database of processed system events. Typically in each case, each kind of system event generates a distinct table, although all event tables will have some features in common (e.g. timestamp, device and/or user identifier).

The approach we have adopted in DRS is to support multiple relational databases internal to each DRS application, as an additional media type. DRS's Log File Workbench component allows new databases to be created dynamically (using HSQLDB) and to be described within a project's RDF model. Other parts of the Log File Workbench support the process of importing system log files, generating corresponding event tables and rows, and viewing and visualising entries in the database.

A large portion of the log file workbench sub-ontology allows the tables and table columns to be described, to identify time stamps, device identifiers, positions, text messages, and so on. This database metadata goes substantially beyond the simple type information available in SQL or used by Replayer or DataGoggles, allowing generic viewers (such as a "smart" table view) to give tailored windows onto the information in a database, alongside existing video and other views.

6.2. XML Files for Editable Documents

One of our user groups has a requirement to author "rich" documents within the DRS environment [15]. Such a document may combine free text, transcription fragments, images, references to particular analyses and to particular times within those analyses, other objects in the metadata (e.g. people), and so on, into a single rich document.

The first implementation of this functionality used a new set of OWL classes to specify the document structure to be persisted in the normal project model. The structure was a list of paragraphs, each comprising a list of spans of text and other items, with text and other style information. While this is technically possible, it is not a particularly good fit with RDF. For example, lists (of paragraphs and text elements) are relatively verbose to define and use in RDF, and the number of text and style elements can grow very quickly in such a document representation, expanding the RDF model very quickly.

Consequently we have redesigned this facility to store the document itself in its own XML schema, persisted as a file on disk (like

the existing video files). These can then be viewed and edited through a variant of the current viewers. The additional complexity which this introduces is that these document files are clearly expected to change – as they are edited – whereas the video and log files are expected to remain unchanged. Consequently some additional metadata and supporting facilities are required to track and handle the evolving versions of these document files. However, the more compact representation and the ability to directly exchange files and work with a more "normal" document format and representation appears to outweigh these factors.

7. Conclusions and Future Work

In general we have found RDF and OWL to be an effective persistent data model for our application. The flexibility of RDF has allowed us to easily extend the data model as required, e.g. to allow provenance information to be associated with resources. However the process of using it has not been particularly simple.

First, performance has been a recurring problem, and a number of strategies have been combined to achieve satisfactory results:

- The RDF model is cached in-memory and updates are only persisted when explicitly requested by the user. This allows the user to manage the delays caused by persistence, and also allows us to avoid the need for duplicate statement checking in the persistent model.
- Very limited use has been made of inference, both the kind of inference (limited RDFS entailments only) and its scope (the ontology only).
- Not all information has been converted to RDF, in particular we also make use of native files for audio, video, source documents and certain editable documents and relational databases for the persistence of structured application event data (for visualisation and replay).

Second, the generic RDF model interface is not very easy (or type safe) to use in many situations. For the software developer this is largely addressed by our use of automatically generated Java wrapper classes for the ontology, which give a Java type-safe interface to the RDF model for simple query and update. However complex queries must still be done via the generic SPARQL interface. A similar issue is seen in the current generic (ontology-driven) user interfaces, which tend to be quite unwieldy and awkward for non-expert users. In specific

cases this is addressed by custom (class-specific) interfaces, such as the project browser. Beyond this we are exploring ways in which the generic interfaces might be refined or tailored (e.g. as in Haystack [17]).

DRS is a work in progress. It is being developed under an open source license (“new” BSD). It is currently available – with a degree of support – directly from the development team, or from the SourceForge project “thedrs” [16]. Source code should be migrated to SourceForge in the next few weeks. The wrappergen tool is also available separately, currently from the project team.

A first major public release is due by the end of August 2007. This will support coordinated replay of multiple video files, textual transcriptions and system logs, plus free text annotation, simple hierarchical coding, and searching. The distributed workgroup facilities are scheduled for a later release, and will be described elsewhere.

8. Acknowledgements

This work was supported by the ESRC through the grant “Understanding New Forms of Digital Record for E-Social Science” (the DReSS node of the NCESS) and by the EPSRC through grant EP/C010078/1, “Semantic Media - Pervasive Annotation for e-Research” and the EQUATOR IRC, grant GR/N15986/01.

9. References

- [1] French, A., Greenhalgh, C., Crabtree, A., Wright, M., Brundell, P., Hampshire, A., and Rodden, T., “Software Replay Tools for Time-Based Social Science Data”, Second International Conference on e-Social Science, 28-30 June 2006, Manchester, UK (available as <http://www.ncess.ac.uk/events/conference/2006/papers/papers/FrenchSoftwareReplayTools.pdf>)
- [2] Fraser, M., Hindmarsh, J., Best, K., Heath, C., Biegel, G., Greenhalgh, C., and Reeves, S. 2006. Remote Collaboration Over Video Data: Towards Real-Time e-Social Science. *Comput. Supported Coop. Work* 15, 4 (Aug. 2006), 257-279. D
- [3] W3C (2006a): ‘Resource Description Framework (RDF)’. <http://www.w3.org/RDF/> (verified 26th April 2007)
- [4] W3C (2006b): ‘Web Ontology Language (OWL)’. <http://www.w3.org/2004/OWL/> (verified 26th April 2007)
- [5] Brian McBride, “Jena: a Semantic Web Toolkit”, IEEE Internet Computing, Nov/Dec 2002, pp. 55-59.
- [6] JENA (2006): ‘A Semantic Web Framework for Java’, <http://jena.sourceforge.net/index.html> (verified 26th April 2007)
- [7] Hypersonic SQL Database, <http://hsqldb.org/> (verified 26th April 2007)
- [8] Priya Parvatikar and Katie Portwin, “Scaling Jena in a commercial environment: The Ingenta MetaStore Project”, 2006 Jena User Conference, May 10th and 11th 2006, Bristol, UK (available as <http://jena.hpl.hp.com/juc2006/proceedings/portwin/paper.pdf>, verified 2007-06-25)
- [9] JENA (2003), “Jena 1 vs Jena 2 performance comparison results”, <http://jena.sourceforge.net/DB/Jena1/perftestWin32.html> (verified 2007-06-24)
- [10] JENA (2003), “Jena2 Database Interface – Performance Notes”, <http://jena.sourceforge.net/DB/Jena1/perfNotes.html> (verified 2007-06-25).
- [11] W3C (2004): ‘RDF Semantics’, W3C Recommendation 10 February 2004, <http://www.w3.org/TR/rdf-nt/> (verified 26th April 2007)
- [12] Protégé, <http://protege.stanford.edu/> (verified 26th April 2007)
- [13] Kazuki, <http://projects.semwebcentral.org/projects/kazuki/> (verified 26th April 2007)
- [14] Crabtree, A., Benford, S., Greenhalgh, C., Tennent, P., Chalmers, M., and Brown, B. 2006. Supporting ethnographic studies of ubiquitous computing in the wild. In *Proceedings of the 6th ACM Conference on Designing interactive Systems* (University Park, PA, USA, June 26 - 28, 2006). DIS '06. ACM Press, New York, NY, 60-69.
- [15] Crabtree, A., French, A., Greenhalgh, C., Benford, S., Cheverst, K., Fitton, D., Rouncefield, M., and Graham, C. 2006. Developing Digital Records: Early Experiences of Record and Replay. *Comput. Supported Coop. Work* 15, 4 (Aug. 2006), 281-319.
- [16] The DRS SourceForge project, <http://sourceforge.net/projects/thedrs> (verified 26th April 2007)
- [17] David Karger, Karun Bakshi, David Huynh, Dennis Quan and Vineet Sinha, “Haystack: A General Purpose Information Management Tool for End Users of Semistructured Data”, Proceedings of CIDR 2005, Jan. 4-7, Asilomar, CA.