

Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF

David Huynh

MIT Artificial Intelligence Laboratory
200 Technology Square
Cambridge, MA 02139
1 (617) 452-5041

dphuynh@ai.mit.edu

David Karger

MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139
1 (617) 258-6167

karger@theory.lcs.mit.edu

Dennis Quan

MIT Artificial Intelligence Laboratory
IBM Internet Technology Division
200 Technology Square
Cambridge, MA 02139
1 (617) 693-4612

dquan@media.mit.edu

ABSTRACT

The Resource Definition Framework (RDF) is designed to support agent communication on the Web, but it is also suitable as a framework for modeling and storing individual users' information. By using RDF in this manner, our Haystack platform provides information management capabilities personalized to individual users. This flexible semi-structured data model is appealing for several reasons. First, RDF supports ontologies created by the user and tailored to the user's needs. At the same time, system ontologies can be specified and evolved to support a variety of high-level functionalities such as flexible organization schemes, semantic querying, and collaboration. In addition, we show that RDF can be used to engineer a user interface architecture that gives rise to a semantically rich and uniform UI. We demonstrate that by aggregating various types of users' data together in a homogeneous representation, we create opportunities for agents to make more informed deductions in automating tasks for users. Finally, we discuss the implementation of an RDF information store and a programming language specifically suited for manipulating RDF.

1. INTRODUCTION

A key present day challenge for any individual is making productive use of the vast amount of information they have and the even vaster amount that they can easily obtain through the Internet. A significant barrier to such use is the heterogeneity of the data itself and of the applications that help users work with that data. Data objects with close semantic relationships, such as an e-mail about a project, a web page relevant to it, a calendar entry of an important project date, and a to-do list about the project cannot be dealt with in a uniform way—they look different and are manipulated by distinct applications that do not talk with each other.

The problem of data heterogeneity on the web is being tackled by the development of RDF, a common language for the description of resources and the relationships connecting them [6]. RDF has been developed to provide interoperability between applications that exchange machine-understandable information on the Web. In other words, RDF is well-suited for facilitating Web Services in resource discovery, cataloging, content rating, and privacy policies.

While it has mainly been targeted at web automation, we argue that RDF has great value as a tool for storing, navigating, and retrieving information in an individual user's information corpus. The data relevant to a particular activity can be brought together

and manipulated by user interfaces optimized for that activity rather than for a particular data type. Common information management tasks, such as searching for information relevant to a particular query or flagging items of particular interest to a user, can be delegated to agents that need not waste time dealing with multiple data formats. By describing all of the user's data with one unified semi-structured data model, Haystack enables the components of the system to concentrate on the semantic differences in the data without worrying about syntactic or protocol differences occurring between different systems. Individual users can extend the data model, adding attributes or schemata that represent their preferred means of organizing information, making it easier for them to find information the way they expect it to be found.

While these added capabilities are attractive, several problems must be solved in order to attain them. Simply storing a user's data in RDF is not sufficient; indeed, it is in some sense a step backward since it is much more complex than the files and folders model with which users currently work. An interface must be developed that takes advantage of RDF's rich structure to give its user a more informative view of their repository without swamping them. We must help people manipulate unstructured, semi-structured, and structured data without requiring them to become skilled database administrators. If agents are to help manipulate this repository, an appropriate environment must be designed within which such agents can architected and can operate and communicate with each other. In this paper, we propose solutions to these problems.

1.1 Motivation

The goal of the Haystack project is to develop a tool that allows users to easily manage their documents, e-mail messages, appointments, tasks, and other information. Haystack is designed to address four specific needs of the user.

First, the user should be allowed maximum flexibility in how he or she chooses to describe and organize his or her information. The system should allow the user to structure his or her data in the most suitable fashion as perceived by the user.

Second, the system should not create artificial distinctions between different types of information that would seem unnatural to the user. This point is related to the previous point in that the system should not partition a corpus simply because different programs are used to manipulate different parts of that corpus. Rather, the system should store all of the user's information in one

homogeneous representation and allow the user to impose semantics that partition the data appropriately.

Third, the system should allow the user to easily manipulate and visualize his or her information in ways appropriate to the task at hand. The user interface should be aware of the context in which arbitrary information is being displayed and should present an appropriate amount of detail.

Fourth, the user should be able to delegate certain information processing tasks to agents. Regardless of how powerful a user interface we provide, there will still be many repetitive tasks facing users, and we feel that users will benefit from automation.

1.2 Contribution

By addressing these four needs, Haystack is able to use its RDF-based semi-structured data model to extend several significant benefits to users. RDF can be readily exploited to add semantics to existing information management frameworks and to serve as a *lingua franca* between different corpora. On top of this, we provide an ontology that supports capabilities including collection-based organization, semantic categorization, collaboration and trust management. By ontology we are referring to a vocabulary that specifies a set of classes and the properties possessed by objects of these classes. A representative screenshot of our system in Section 2 helps to illustrate some of these capabilities further.

In Section 3, we argue that representing information in RDF lets us develop a **semantic user interface** for presenting information in a uniform, natural fashion. Instead of presenting each data type inside a distinct application, our interface allows distinct data types to be seamlessly presented together, with appropriate views of each type of data being integrated and nested at an extremely fine level of detail, and appropriate actions for each data type being available whenever data of that type is in view. In a convenient self-reference, RDF itself provides a natural way to represent these information views, allowing the system to incorporate new “view types” as they are needed.

In Section 4, we discuss the use of RDF for modeling imperative computational processes. We describe an environment within which agents can act to automate information management tasks for the user. We present a language called Adenine as a natural means for manipulating metadata and thus writing programs for Haystack. Adenine programs compile into an RDF representation, affording them the same ability to be annotated, distributed, and customized as other documents and information. This language lowers the barrier for writing user interface components and agents in our system that must flexibly adapt to the underlying data model.

1.3 History

The information overload problem has become more and more evident in the past decade, driving the need for better information management tools. Several research projects have been initiated to address this issue. The Haystack project [9] [10] was started in 1997 to investigate possible solutions to this very problem. It aims to create a powerful platform for information management. Since its creation, the project has sought a data modeling framework suitable for storing and manipulating a heterogeneous corpus of metadata in parallel with a user’s documents. With the introduction of RDF, a good match was found between the versatility and expressiveness of RDF and the primary need of

Haystack to manage metadata. The project has recently been reincarnated to make use of RDF as its primary data model.

1.4 Related Work

There have been numerous efforts to augment the user’s data with metadata. The Placeless Documents project at Xerox PARC [3] developed an architecture for storing documents based on properties specified by the user and by the system. Like Haystack, Placeless Documents supported arbitrary properties on objects and a collection mechanism for aggregating documents. It also specified in its schema access control attributes and shared properties useful for collaboration. The Placeless Documents architecture leveraged existing storage infrastructure (e.g. web servers, file systems, databases, IMAP, etc.) through a driver layer. Similarly, Haystack takes advantage of the same storage infrastructure, using URLs to identify documents.

While it may seem that Placeless and Haystack are quite alike in that they use similar data models, much more significant difference appears in the approach to the user interface. Placeless’ Presto user interface focused on facilitating management of data in general using a predetermined set of interfaces. In developing Haystack, we are experimenting with ways to incorporate the customization of user interfaces into the bigger problem of personalized information management by providing a platform upon which user interfaces can be modeled and manipulated with the same facility as other metadata.

There are other systems, many in common use today, that permit arbitrary metadata annotations on files. The Windows NT file system (NTFS) supports file system-level user-definable attributes. WebDAV [2], a distributed HTTP-based content management system, also permits attributes on documents. Lotus Notes and Microsoft Exchange, two common knowledge management server solutions, both support custom attributes on objects within their databases. However, the metadata are not readily interchangeable among different environments. The WebDAV and NTFS metadata formats offer little information on how to present such metadata to the user, making it difficult to construct user interfaces that improve upon standard key/value pair editors. Lotus Notes and Microsoft Exchange store form designs in their stores, but these designs are highly coupled to the specific schemata of the databases they present and must be retrofitted when these schemata are customized. Further, the structure of metadata in these systems is highly constrained and makes the expression of complex relationships between objects difficult. For example, these systems do not have first class support for making assertions about predicates, such as the fact that a digital signature is not a useful property to display to the user in raw form, making it difficult for the user interface and agents to analyze data conforming to a foreign ontology dynamically.

The Semantic Web project at the World Wide Web Consortium (W3C), like Haystack, is using RDF to address these issues of interchangeability [4]. The focus of the Semantic Web effort is to proliferate RDF-formatted metadata throughout the Internet in much the same fashion that HTML has been proliferated by the popularity of web browsers. By building agents that are capable of consuming RDF, data from multiple sources can be combined in ways that are presently impractical. The simplest examples involve resolving scheduling problems between different systems running different calendaring servers but both speaking RDF. A

more complex example is one where a potential car buyer can make automated comparisons of different cars showcased on vendors' web sites because the car data is in RDF. Haystack is designed to work within the framework of the Semantic Web. However, the focus is on aggregating data from users' lives as well as from the Semantic Web into a personalized repository.

2. OVERVIEW OF THE SYSTEM

In order to motivate the types of problems that need to be overcome in a system that aggregates information of various types from various sources, we illustrate an example interaction between the user and Haystack.

Figure 1 shows the user's home page, which is displayed when Haystack's user interface (called Ozone) is first started. Like a portal, the Ozone home page brings together in one screen information important to the user. This information is maintained by agents working in the background. The actual presentation of this information is **decoupled** from the agents and is the responsibility of Ozone UI components called **views**. For instance, the home page displays the user's Items Needing Attention collection, which is managed by the Incoming Agent. When

messages arrive, the Incoming Agent may decide to enter them into the Items Needing Attention collection. Similarly, when read messages have been in the Items Needing Attention collection for some period of time, the Incoming Agent may decide to remove them. These mutations to the Items Needing Attention collection are automatically detected by the collection view sitting on the home page; the view updates the display accordingly. One can envision the Incoming Agent taking on more intelligent behaviors in the future, such as moving a message deduced to be important but yet unread to the top of the collection.

The Items Needing Attention collection, like an e-mail inbox, displays a list of e-mail messages that were recently received and need to be read by the user. However, like all collections in Haystack, the Items Needing Attention collection is **heterogeneous** and contains meetings, to-do items, and other documents in addition to e-mail messages.

When a view of a collection is rendered to the screen, Ozone iterates through the members of the collection, recursively locates views for these members, and instantiates them within the collection view. In other words, Ozone calls upon views

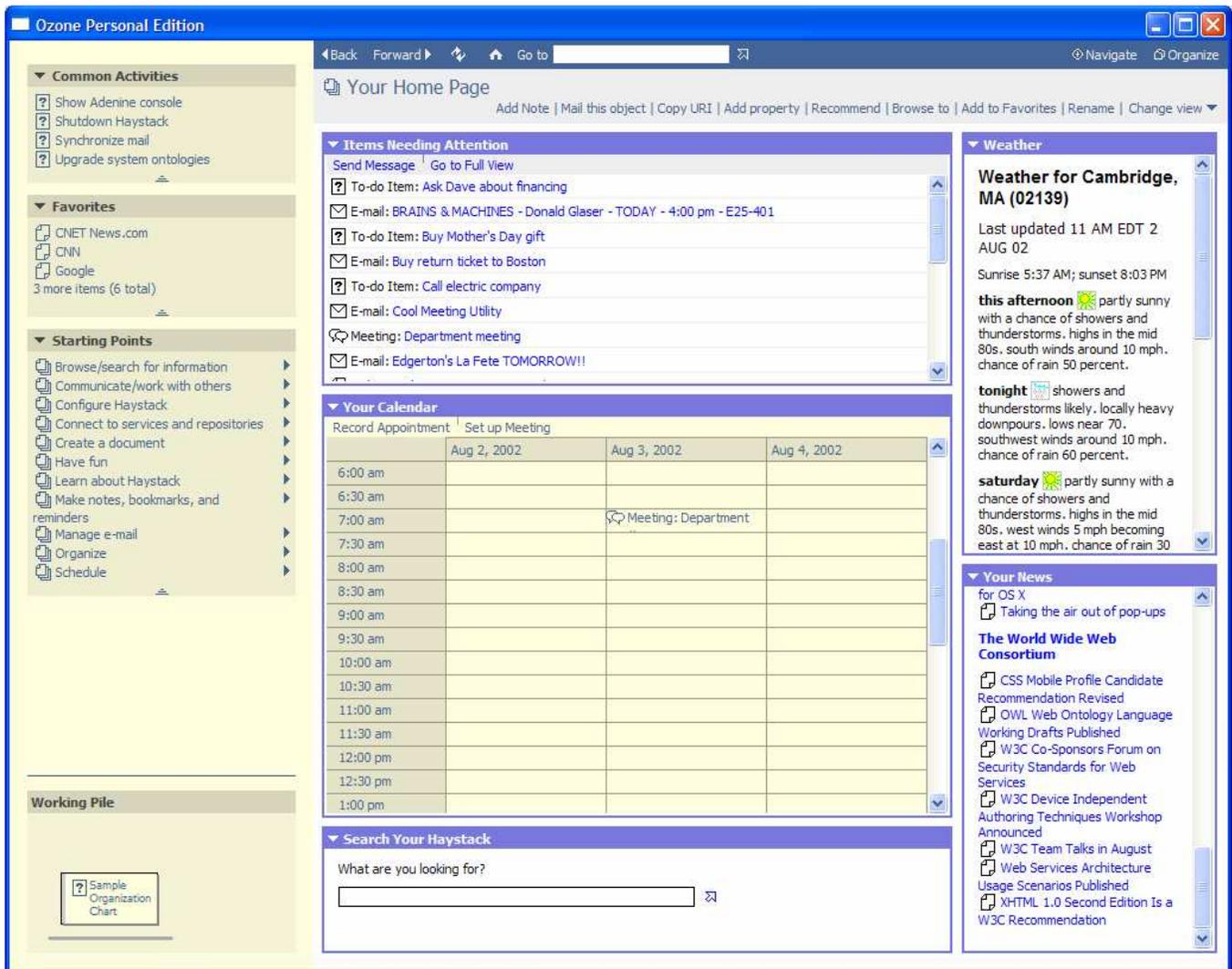


Figure 1. Ozone screenshot

specifically designed to present certain types of data to display those data.

Furthermore, strings of text on the screen corresponding to meetings, to-do items, or e-mail messages are not merely dead pixels. Instead, users can manipulate them with **context menus** and **drag and drop** them between different areas of the screen. For example, one can imagine dragging an e-mail from the Items Needing Attention view to the calendar in order to set up an appointment. Because the UI framework maintains the mapping from each visual UI element, e.g. strings of text, to associated underlying semantic objects, e.g. e-mail messages, the drag and drop operation can be associated with the semantic object being dragged, and as a result, the calendar view can intelligently determine an appropriate response to the drop operation. This mapping forms the basis of what we call the **Semantic User Interface**.

By removing the burden of user interface rendering from the agents, the software designers are encouraged to enrich the agents with more capabilities. One can imagine prolific collaboration between different agents in the Haystack system. For instance, upon retrieving the weather forecast for today, the Weather Agent can notify the Calendar Agent of the grave possibility of a snow storm approaching; the Calendar Agent in turn can attempt to reschedule the user's appointments appropriately. In other systems, especially portals, the focus of a weather agent would be on rendering the weather as HTML, not interacting with other agents to maximize end user benefit.

The news applet displays news downloaded from Resource Site Summary (RSS) feeds of interest to the user. The RSS Agent downloads news on a periodic basis and incorporates the RSS files (which are RDF) into the user's corpus. To take advantage of the collection view for displaying news, another agent translates the news from the RSS ontology into the Haystack collection ontology. In the future it will be possible to have another agent filter the RSS feeds for the particular articles thought to be most interesting to the user.

Furthermore, the layout of the entire home page and all of its customizations are described in metadata. As with other objects, this layout can be annotated, categorized, and sent to others.

3. SUPPORTING USER INTERACTION WITH SEMI-STRUCTURED DATA

As Haystack uses RDF to bring together various kinds of information from different sources, there is a need to present and allow the user to interact with such a diverse pool of data in a systematic and manageable fashion. In existing software applications, the popular way, if not the only way, to deal with heterogeneous information is to segregate the data into different storage formats and to allow access to it through separate independent programs. The segregation by formats locks up the data in the various applications and prevents rich aggregations of the information that are useful and meaningful to the user. Such rich aggregations are also rendered impossible when each type of data is only accessible through the interface of one application. In other words, the user interfaces of existing applications cannot cooperate or mingle.

Now that Haystack's use of RDF has removed the segregation of data by formats, there arises an opportunity to create a new user interface that provides seamlessly combined presentations of

previously disparate information. We propose the Semantic User Interface (SUI) paradigm, which relies on extensive reuse of UI components to present heterogeneous information. In this paradigm, every piece of information, however small, is assigned one or more UI components called **views** capable of rendering it. Larger bodies of information are rendered by assembling together the views of smaller pieces. In such a fashion Haystack's entire UI is constructed dynamically.

We propose that the SUI paradigm has several benefits. First, the extensive reuse of UI components avoids duplicate work for the UI designers and lowers the barrier of UI work for non-UI designers. Second, the paradigm also dictates that views be kept in synchrony with the pieces of data that they display, allowing one to change the data being presented without having to actively update the presentation. Consequently, one can concentrate on processing information without having to worry about how that information is being displayed. Finally, extensive reuse of UI components and systematic UI construction result in a more consistent and predictable user interface for the user to interact with.

3.1 The Problem with Existing UI Technologies

Before we dive into the principles of the Semantic UI paradigm and its implementation details in Haystack, it is crucial to point out the major flaw of existing UI technologies that render them unsuitable for the needs of our system: they rarely allow UI work to be reusable. Each UI designer must add his or her own code to present data that is already displayed in proper form somewhere else by someone else. Take the example in which a UI designer needs to develop the interface for a CD burning utility that allows the user to select several files from several directories to transfer to a CD. The selected files should preferably be displayed in a list view with useful fields including file name, file size, file type, date, author, etc. The operating system's file browser already knows how to render a list view showing files in a single directory. In fact, the file browser knows how to render a list view item for any file in any directory. However, it is not possible to embed such list view items rendered by the file browser to show files from different directories in a single list view. The granularity of embeddable components is at the list view level, not at the list view item level. Consequently, the UI designer is forced to attempt to duplicate some rendering capabilities of the file browser in his or her own software.

There do exist some mechanisms for embedding UI components within one another (e.g., Object Linking & Embedding (OLE) framework, Model-View-Controller (MVC) paradigm), but these mechanisms are insufficient for our purposes. In the MVC paradigm, a UI designer wishing to reuse an existing view must explicitly specify that view's implementation in his or her UI construction code. Should the view's implementation be replaced, the UI designer's code becomes outdated. The OLE framework resolves this problem by providing a dynamic binding scheme that looks up and then embeds view implementations dynamically at runtime. However, for each piece of data whose view is desired, the OLE framework can only instantiate one view—often the content view. The UI designer cannot specify the type of view to embed. In many cases, the view provided for a piece of data is too coarse in the context where it is embedded. In the example of the CD burning utility, individual list view items are needed but only a whole list view can be reused.

Because existing UI technologies are not powerful enough to support flexible reuse of UI components, each UI designer is left to improvise his or her own user interface design. He or she writes code to display almost each and every type of data that his or her application deals with. Even in the same application, features written by different UI designers contain different code fragments to display the same type of data. Those code fragments provide different UI capabilities to their corresponding UI elements. In many cases, the piece of data that the user wants to interact with is readily displayed, but its UI representation is no more than dead pixels on the screen, affording no means for interaction, so that the user is forced to take a different UI route to manipulate it. This is a subtle but prevalent and severe inconsistency in today's user interfaces.

Figure 2 and Figure 3 illustrate such an inconsistency. A contact name shown in the "From" column for an e-mail message in a list view (Figure 2) should expose the same actions as the same contact name shown in the "From" text field in the e-mail message window (Figure 3). In Microsoft Outlook 2002 and other existing e-mail clients, those two elements provide almost entirely different sets of actions. The former element is a dead text string painted as part of the whole list view item representing the e-mail message. Right-clicking on it is equivalent to right-clicking anywhere else in that list view item. The same context menu for the whole message is always shown regardless of the right-click location. The latter element is a control by itself. It represents a contact object and shows the context menu applicable to that object when right-clicked. To the user, both elements represent the same contact object and should give the same context menu. It is this type of inconsistency that we wish to eliminate in Haystack's user interface.

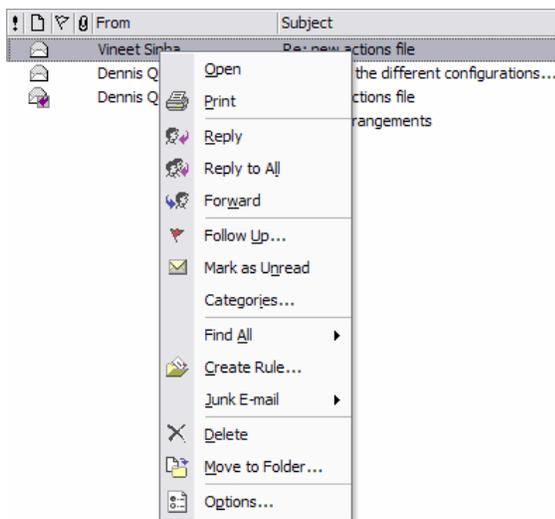


Figure 2. Actions for a contact name in a list view (Microsoft Outlook 2002)

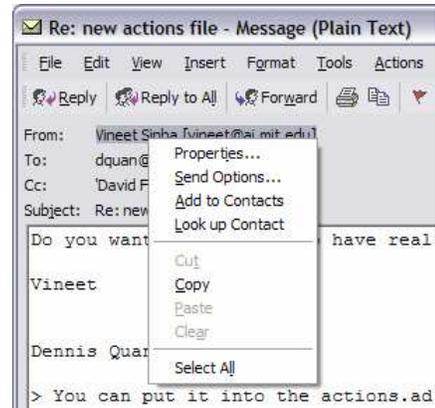


Figure 3. Actions for a contact name in the e-mail compose window (Microsoft Outlook 2002)

3.2 Principle of Reuse

In order to build a consistent user interface, the Semantic User Interface paradigm facilitates and encourages extensive reuse of UI components. The paradigm specifies a dynamic binding scheme for embedding views much like the scheme used in the OLE paradigm. However, while OLE and MVC rarely employ more than one level of embedding, the SUI paradigm encourages arbitrarily deep nesting of views. Furthermore, SUI views need not be rectangular child windows. They can be inline segments of text that flow through several lines. This flexibility makes views versatile and easy to embed anywhere. In fact, the SUI paradigm strongly advocates that each UI designer specializes in handling only the types of data that he or she knows best and embeds views made by other UI designers for other types of data.

The mapping from a piece of information, or a type of information, to the views capable of rendering it is stored entirely as metadata in the RDF store of Haystack. The metadata describes the types and formats of data that each view is capable of presenting as well as the contexts in which each view is appropriate. Note that each piece of information can have more than one view: an audio file can be summarized in one line of text based on its play time, artist, title, etc., or it can be viewed in an audio player that takes up a whole window. The former view is appropriate where a short description is desired, and the latter should be used when the user focuses solely on the audio file. Just as the data types to be displayed and their schemata change frequently, so do the metadata for the views capable of rendering them. RDF is powerful enough to capture all of such view mapping specifications and to handle all future extensions to the system. Furthermore, RDF's generic benefit of portability allows for easy deployment and upgrade of UI components and easy exchange of UI capabilities among Haystack users.

3.3 Constructing UI Dynamically

The Haystack user interface infrastructure provides a component called the **view selector** that performs the mapping from data to view automatically. While designing a particular view, a UI designer can insert view selectors to embed inner views within this view. At runtime, the view selectors look up and instantiate appropriate inner views. In essence, the view selectors compose the UI dynamically as a hierarchy of nested views by acting as the level of indirection in the dynamic binding scheme of the SUI paradigm. They effectively raise the level of abstraction for the UI

designers, inasmuch as the designer of the outer view needs not know the details of how the inner view is constructed but can simply delegate the task of constructing the proper inner view to the view selector.

Figure 4 shows an example of nested views: part (b) highlights the boundaries of the view selectors used to construct the piece of UI shown in part (a). The view for a meeting is built by embedding the views for the various meeting fields such as Time, Location, and Attendees. The value of the Location field is a Room object, which is rendered with a view. This view in turn embeds another view that renders the building “B13.” Similarly, the view showing the collection of attendees embeds an inner view for each of the collection members. Note that the view for the collection is not rectangular: it flows like text onto more than one line.

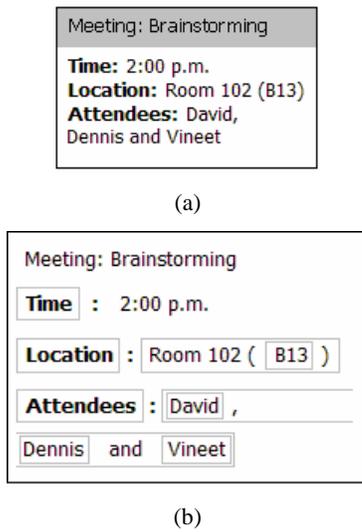


Figure 4. Example of nested views

When the view selector component is used as the sole mechanism for constructing views, it becomes a single point of intelligence. Its capabilities are inherited in every part of the UI and any improvement to the component translates to a system-wide improvement.

3.4 Keeping UI Up to Date

In order to make a view a faithful representation of the corresponding piece of information, the SUI paradigm specifies that the view should register for notifications from the RDF store upon any change to the information that it displays. For instance, “stacker” views have been designed to present collections dynamically. (A collection is a mathematical set of objects.) Given a collection, a stacker view constructs a view selector for each element in that collection and stacks the view selectors in some specified sorting order. The stacker view also registers for notifications upon any change to that collection. If a new element is added to the collection, the stacker view constructs a new view selector for it. If an existing element is removed, the stacker view removes the corresponding view selector. Figure 1 shows several stacker views used to display collections including “Favorites” and “Your News.”

A UI designer who makes use of a stacker view needs only specify the sorting order for the elements and the specifications

for the dynamically constructed view selectors so that appropriate views of elements are produced. The stacker views have effectively raised the level of abstraction for rendering collections. In the future, grouping and other high level presentation logics will be supported.

The stacker views, with their ability to update the UI dynamically, effectively decouple the information processing tasks from the UI presentation tasks. That is, one can concentrate on managing the elements inside a collection without concerning oneself with how that collection is being displayed. Since the UI gives a faithful representation of the data, the view of that collection always reflects the contents of the collection. This capability lowers the barrier of entry for development for both UI designers and those concerned solely with information processing.

3.5 Supporting UI Features Uniformly

Because each piece of data is presented by a view, for any pixel on the screen, there is a systematic way to detect which of the currently displayed views enclose that pixel and to trace back to the corresponding pieces of data being presented by those views. That is, for every rendered pixel there is a connection back to the data that the pixel represents. For this reason the paradigm is called “semantic.”

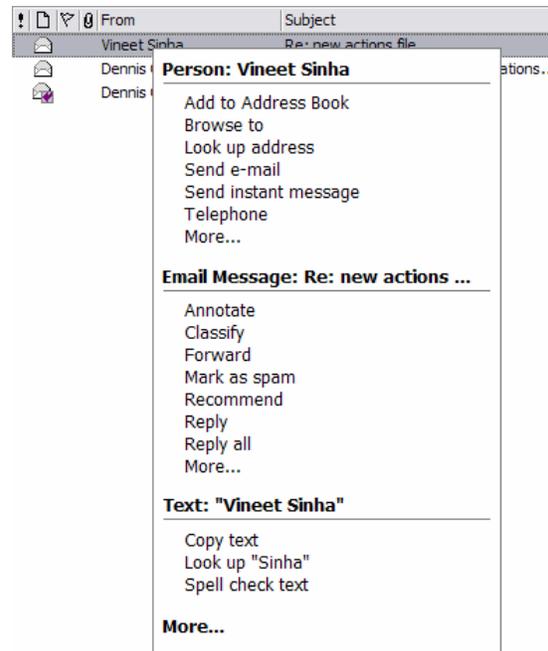


Figure 5. Sample context menu elements presenting the same piece of data afford the same set of actions corresponding to that data.

Using these live connections between elements on the screen and the data they represent, it is easy to systematically support features such as context menus and drag and drop consistently throughout the Haystack user interface. For instance, in Figure 5, when the user right-clicks on the text string “Vineet Sinha”, we can trace back through all of the pieces of data that the clicked pixel represents and construct a context menu listing all actions applicable to those pieces of data. Of note in Figure 5 are some capabilities not provided in existing e-mail clients. In particular,

the “Text” section in the menu offers ways to copy, look up, and spell-check an otherwise dead piece of text. In existing e-mail clients, only text inside e-mail bodies can be spell-checked. One can also imagine the usefulness of spell-checking file names [11] and e-mail subjects. Systematic UI construction in our system yields a uniform user interface in which features such as context menus and drag and drop can be provided pervasively throughout the whole application, making the application behave consistently and predictably to the user.

4. AGENT INFRASTRUCTURE

We now turn our attention to agents, which play an important role in not only improving the user experience with regards to keeping information organized, but also in performing tedious tasks or well-defined processes for the user. We also describe some underlying infrastructure needed to make writing and using agents in Haystack efficient and secure.

4.1 Agents

In the past, programs that aggregated data from multiple sources, such as mail merge or customer relationship management, had to be capable of speaking numerous protocols with different backends to generate their results. With a rich corpus of information such as that present in a user’s Haystack, the possibility for automation becomes significant because agents can now be written against a single unified abstraction. Furthermore, agents can be written to help users deal with information overload by extracting key information from e-mail messages and other documents and presenting the user with summaries.

As we alluded to earlier, collections can be maintained automatically by agents. Modern information retrieval algorithms are capable of grouping documents by similarity or other metrics, and previous work has found these automatic classifications to be useful in many situations [14]. Additionally, users can build collections prescriptively by making a query. An agent, armed with a specification of what a user is looking for, can create a collection from the results of a query, and it can watch for new data entering the system that matches the query.

For example, agents can automatically filter a user’s e-mail for documents that appear to fit in one or more collections defined by the user, such as “Website Project” or “Letters from Mom”. Because membership in collections is not one-to-one, this classification can occur even while the message remains in the user’s inbox.

Agents are used in Haystack to automatically retrieve and process information from various sources, such as e-mail, calendars, the World Wide Web, etc. Haystack includes agents that retrieve e-mail from POP3 servers, extract plaintext from HTML pages, generate text summaries, perform text-based classification, download RSS subscriptions on a regular basis, fulfill queries, and interface with the file system and LDAP servers.

The core agents are mostly written in Java, but some are written in Python. We utilize an RDF ontology derived from WSDL [5] for describing the interfaces to agents as well as for noting which server processes hosts which agents. As a consequence, we are able to support different protocols for communicating between agents, from simply passing in-process Java objects around to using HTTP-based RPC mechanisms such as HTTP POST and SOAP [1].

4.2 Belief

When multiple agents are used to generate the same information, issues arise as to how to deal with conflicts. For instance, if one agent is tasked with determining the due date of a document by using natural language processing and another agent does the same by extracting the first date from a document, which is to be believed when there is a conflict? In instances such as this, it is important that information be tagged with authorship metadata so the user can make an informed choice of which statement to choose.

To accomplish this we discuss a part of the system ontology that is used for describing attributes about actual statements themselves, such as who asserted them and when they were asserted. Under the premise that only three values, namely subject, predicate, and object, are required to describe statements in our model, it is possible to give statements identifiers and to assert an author and creation time to the original statement. In fact, the RDF model prescribes that in order to make statements about statements, the referent statement must be reified into a resource and assigned a URI, and the referring statements can then use the reified resource in the subject or object field.

This use of reification brings up a subtle issue concerning RDF. In a document containing RDF, it is assumed that all statements are asserted to be true by the author [8]. In order to make a statement about another statement that the author does not necessarily believe is true, the target statement must exist only in reified form. In essence, the author is binding a name to a specific statement with a certain subject, predicate, and object, but is not asserting the statement to be true, only instead asserting other properties about that statement using the name.

Keeping track of various levels of trustworthiness is important in a system that contains statements made by numerous independent agents, as well as information from users’ colleagues, friends, family, solicitors, and clients. In order to maintain metadata on the statements themselves in an information store, one solution is to have the information store become a “neutral party”, recording who said what and when those things were said, but not asserting their truth. This is accomplished by having all statements made by parties other than the information store reified. (An alternative is to have one entity—perhaps the user—be at the same trust level as the data store. However, this results in statements made by the user being handled in one fashion and those made by others (which have been reified) handled in a different fashion. For simplicity of implementation, we keep the data store neutral.)

Once we have a system for recording statement metadata, we can examine issues of retraction, denial, and expiration of assertions, *i.e.*, statements asserted by specific parties. Consider an example where an agent is responsible for generating the title property for web pages. Some web pages, such as those whose contents are updated daily, have titles that change constantly. Often users want to be able to locate pages based on whatever it is they remember about the page. One approach for handling constant mutations in the information store is to allow agents to delete a previous assertion (*i.e.*, remove the statement from the database) and to replace it with an up-to-date version. However, it would be powerful to allow users to make queries of the form “Show me all web pages that had the title *Tips for Maintaining Your Car* at some point in time.” By allowing agents to retract their assertions, that is, record that an agent no longer asserts the statement without

removing it, queries can still be made to retrieve past or obsolete information because this information is not deleted. Additionally, this system permits users to override an assertion made by an agent by denying the assertion, yet retains the denied assertion for future reference.

In a system such as this where multiple parties and agents provide information, we are often concerned with impersonation and forgery. To solve these problems, we propose supporting digitally signed RDF. The digital signature permits the information store to determine and verify the author of statements with certainty. In an ideal system, users and agents sign all RDF they produce with assigned digital signatures. However, the W3C is still working on the details of supporting signed RDF at the statement level, and the implementation of a digital signature system is beyond the scope of this project. For our current prototype, identifier strings are used in place of true signatures.

4.3 Adenine

In a system such as Haystack, a sizeable amount of code is devoted to creation and manipulation of RDF-encoded metadata. We observed early on that the development of a language that facilitated the types of operations we frequently perform with RDF would greatly increase our productivity. As a result, we have created Adenine. An example snippet of Adenine code is given in Error! Reference source not found..

The motivation for creating this language is twofold. The first key feature is making the language's syntax support the data model. Introducing the RDF data model into a standard object-oriented language is fairly straightforward; after all, object-oriented languages were designed specifically to be extensible in this fashion. Normally, one creates a class library to support the required objects. However, more advanced manipulation paradigms specific to an object model begin to tax the syntax of the language. In languages such as C++, C#, and Python, operator overloading allows programmers to reuse built-in operators for manipulating objects, but one is restricted to the existing syntax of the language; one cannot easily construct new syntactic structures. In Java, operator overloading is not supported, and this results in verbose APIs being created for any object oriented system.

Arguably, this verbosity can be said to improve the readability of code. On the other hand, lack of syntactic support for a specific object model can be a hindrance to rapid development. Programs can end up being much long than necessary because of the verbose syntactic structures used. This is the reason behind the popularity of domain-specific programming languages, such as those used in Matlab, Macromedia Director, etc. Adenine is such a language. It includes native support for RDF data types and makes it easy to interact with RDF containers and services.

The other key feature of Adenine is its ability to be compiled into RDF. The benefits of this capability can be classified as portability and extensibility. Since 1996, p-code virtual machine execution models have resurged as a result of Java's popularity. Their key benefit has been portability, enabling interpretation of software written for these platforms on vastly different computing environments. In essence, p-code is a set of instructions written to a portable, predetermined, and byte-encoded ontology.

```
# Prefixes for simplifying input of URIs
@prefix : <urn:test-namespace:>

:ImportantMethod rdf:type rdfs:Class

method :expandDerivedClasses ; \
rdf:type :ImportantMethod ; \
rdfs:comment \
"x rdf:type y, y rdfs:subClassOf z => x rdf:type z"
# Perform query
# First parameter is the query specification
# Second is a list of the variables to return,
# in order
= data (query {
?x rdf:type ?y
?y rdfs:subClassOf ?z
} (List ?x ?z))

# Assert base class types
for x in data
# Here, x[0] refers to ?x
# and x[1] refers to ?z
add { x[0] rdf:type x[1] }
```

Figure 6. Sample Adenine code

Adenine takes the p-code concept one step further by making the ontology explicit and extensible and by replacing byte codes with RDF. Instead of dealing with the syntactic issue of introducing byte codes for new instructions and semantics, Adenine takes advantage of RDF's ability to extend the directed "object code" graph with new predicate types. One recent example of a system that uses metadata-extensible languages is Microsoft's Common Language Runtime (CLR). In a language such as C#, developer-defined attributes can be placed on methods, classes, and fields to declare metadata ranging from thread safety to serializability. Compare this to Java, where serializability was introduced only through the creation of a new keyword called `transient`. The keyword approach requires knowledge of these extensions by the compiler; the attributes approach delegates this knowledge to the runtime and makes the language truly extensible. In Adenine, RDF assertions can be applied to any statement.

These two features make Adenine very similar to Lisp, in that both support open-ended data models and both blur the distinction between data and code. However, there are some significant differences. The most superficial difference is that Adenine's syntax and semantics are especially well-suited to manipulating RDF data. Adenine is mostly statically scoped, but has dynamic variables that address the current RDF containers from which existing statements are queried and to which new statements are written. Adenine's runtime model is also better adapted to being run off of an RDF container. Unlike most modern languages, Adenine supports two types of program state: in-memory, as is with most programming languages, and RDF container-based. Adenine in effect supports two kinds of closures, one being an in-memory closure as is in Lisp, and the other being persistent in an RDF container. This affords the developer more explicit control over the persistence model for Adenine programs and makes it possible for agents written in Adenine to be distributed.

The syntax of Adenine resembles a combination of Python and Lisp, whereas the data types resemble Notation3 [12]. As in Python, tabs denote lexical block structure. Backslashes indicate a continuation of the current line onto the next line. Curly braces ({}) surround sets of RDF statements, and identifiers can use namespace prefixes (e.g. `rdf:type`) as shorthand for entering full URIs, which are encoded within angle brackets (<>). Literals are enclosed within double quotes.

Adenine is an imperative language, and as such contains standard constructs such as functions, for loops, arrays, and objects. Function calls resemble Lisp syntax in that they are enclosed in parentheses and do not use commas to separate parameters. Arrays are indexed with square brackets as they are in Python or Java. Also, because the Adenine interpreter is written in Java, Adenine code can call methods and access fields of Java objects using the dot operator, as is done in Java or Python. The execution model is quite similar to that of Java and Python in that an in-memory environment is used to store variables; in particular, execution state is *not* represented in RDF. Values in Adenine are represented as Java objects in the underlying system.

Adenine methods are functions that are named by URI and are compiled into RDF. To execute these functions, the Adenine interpreter is passed the URI of the method to be run and the parameters to pass to it. The interpreter then constructs an initial in-memory environment binding standard names to built-in functions and executes the code one instruction at a time. Because methods are simply resources of type `adenine:Method`, one can also specify other metadata for methods. In the example given, an `rdfs:comment` is declared and the method is given an additional type, and these assertions will be entered directly into the RDF container that receives the compiled Adenine code.

The top level of an Adenine file is used for data and method declarations and cannot contain executable code. This is because Adenine is in essence an alternate syntax for RDF. Within method declarations, however, is code that is compiled into RDF; hence, methods are like syntactic sugar for the equivalent Adenine RDF “bytecode”.

Development on Adenine is ongoing, and Adenine is being used as a platform for testing new ideas in writing RDF-manipulating agents.

5. DATA STORAGE

5.1 RDF Store

Throughout this paper we have emphasized the notion of storing and describing all metadata in RDF. It is the job of the RDF store to manage this metadata. We provide two implementations of the RDF store in Haystack. The first is one built on top of a conventional relational database utilizing a JDBC interface. We have adopted HSQL, an in-process JDBC-compatible database written in Java. However, early experiments showed that for the small but frequent queries we were performing to render Ozone user interfaces (e.g., following one or two edges from a node), the system was slowed by the fixed marshalling and query parsing costs involved in composing and then decomposing plaintext SQL queries. (The home page displayed in Figure 1 requires well over 45,000 queries to render.) Switching to a larger scale commercial database appears to result in worse performance because of the socket connection layer that is added in the process.

To solve these problems we developed an in-process RDF database written in C++ (we use JNI to connect it to the rest of our Java code base). By making it specifically suited to RDF, we were able to optimize the most heavily used features of the RDF store while eliminating a lot of the marshalling and parsing costs. However, we acknowledge this to be a temporary solution, and in the long term we would prefer to find a database that is well-suited to the types of small queries that Haystack performs.

5.2 Storing Unstructured Content

It is important for us to address how Haystack interacts with unstructured data in the existing world. Today, URLs are used to represent files, documents, images, web pages, newsgroup messages, and other content accessible on a file system or over the World Wide Web. The infrastructure for supporting distributed storage has been highly developed over the past decades. With the advent of technologies such as XML Namespaces and RDF, a larger class of identifiers called URIs subsumed URLs. Initially, RDF provided a means for annotating web content. Web pages, identified by URL, could be referred to in RDF statements in the subject field, and this connected the metadata given in RDF to the content retrievable by the URL. This is a powerful notion because it makes use of the existing storage infrastructure.

However, with more and more content being described in RDF, the question naturally arises: why not store content in RDF? While this is certainly possible by our initial assumption that RDF can describe anything, we argue this is not the best solution for a couple of reasons. First, storing content in RDF would be incompatible with existing infrastructure. Second, leveraging existing infrastructure is more efficient; in particular, using file I/O and web protocols to retrieve files is more efficient than using XML encoding.

Hence, we do not require that existing unstructured content be stored as RDF. On the contrary, we believe it makes sense to store some of the user’s unstructured data using existing technology. In our prototype, we provide storage providers based on HTTP 1.1 and standard file I/O. This means that storing the content of a resource in Haystack can be performed with HTTP PUT, and retrieving the content of a resource can be performed with HTTP GET, analogously to how other resources’ contents (e.g., web pages) are retrieved. Our ontology uses the `Content` class and its derivatives, `HTTPContent`, `FilesystemContent`, and `LiteralContent` to abstract the storage of unstructured information.

6. FUTURE WORK

Haystack provides a powerful platform for organizing and manipulating users’ information. In this section we touch upon two topics we are currently investigating that build new abstractions on top of the data model discussed above.

6.1 Collaboration

Enabling users to work together, exchange information, and communicate has become an absolutely essential feature of modern information management tools. The focus of current off-the-shelf products has been on e-mail and newsgroup-style discussions. However, the addition of rich metadata manipulation facilities creates many possibilities for Haystack in fostering collaboration.

First, Haystack encourages users to have individualized ontologies, so converting between these ontologies when exchanging data will need to be examined. Agents can be instructed in the relationships between different ontologies and can perform conversion automatically. As an alternative one can imagine an ontological search engine that is consulted whenever a user enters data. This way users end up using the same ontologies to describe similarly-structured data.

Second, security issues arise when sharing data. Support for belief networks will need to be expanded to allow users to distinguish their own information from information obtained from others.

Access control and privacy will need to be examined to allow users to feel comfortable about storing information in Haystack.

Finally, metadata describing individual users' preferences towards certain topics and documents can be used and exchanged to enable collaborative filtering. Sites such as epinions.com promote user feedback and subjective analysis of merchandise, publications, and web sites. Instead of going to a separate site, users' Haystacks can aggregate this data and, by utilizing the belief network, present users with suggestions.

6.2 Customizing Organization Schemes

We have started to investigate the many ways in which people organize their personal information in physical form, such as bookcases and piles. We believe that each method of organization has different advantages and disadvantages in various situations. In light of this, we propose to support several virtual organization schemes simultaneously, such that the user can choose the appropriate organization scheme to use in each situation. Different schemes act like different lenses on the same corpus of information. We will provide agents that help the user create and maintain these organization schemes.

6.3 Customizing Schemata

We are looking to integrate features that allow users to create ontologies implicitly as they input information into the system. For example, when manipulating data in a graph display such as a diagram editor, users are creating terms in an ontology when they define custom relationships between different nodes in the graph. There is also a need to be able to associate terms defined by the user with terms defined by other users. We speculate that agents will be able to unify terms together in the background once connections are discovered.

6.4 Customizing User Interfaces

The fact that the user interfaces are themselves described in RDF means that at some level modifying the user interface can be seen as a typical metadata manipulation task not that different from, say, managing a collection. We aim to develop tools that will make it easy for the user to select and customize the most effective views for the information with which they work.

7. ACKNOWLEDGEMENTS

This work was supported by the MIT-NTT collaboration, the MIT Oxygen project, a Packard Foundation fellowship, and IBM. The authors would like to thank Vineet Sinha for his insight regarding the ideas discussed in this paper.

8. REFERENCES

- [1] Box, D., Ehnebuske, D., Kavivaya, G., et al. *SOAP: Simple Object Access Protocol*.
<http://msdn.microsoft.com/library/en-us/dnsoasps/html/soaspec.asp>.
- [2] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D. *HTTP Extensions for Distributed Authoring – WEBDAV*.
<http://asg.web.cmu.edu/rfc/rfc2518.html>.
- [3] Dourish, P., Edwards, W.K., et al. "Extending Document Management Systems with User-Specific Active Properties." *ACM Transactions on Information Systems*, vol. 18, no. 2, April 2000, pages 140–170.
- [4] Berners-Lee, T., Hendler, J., and Lassila, O. "The Semantic Web." *Scientific American*, May 2001.
- [5] Christensen, E., Cubera, F., Meredith, G., and Weerawarana, S. *Web Services Description Language (WSDL) 1.1*.
<http://www.w3.org/TR/wsdl>.
- [6] *Resource Description Framework (RDF) Model and Syntax Specification*. <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>.
- [7] *Resource Description Framework (RDF) Schema Specification*. <http://www.w3.org/TR/1998/WD-rdf-schema/>.
- [8] *RDF Model Theory*. <http://www.w3.org/TR/rdf-mt/>.
- [9] Adar, E., Karger, D.R., and Stein, L. "Haystack: Per-User Information Environments" in *1999 Conference on Information and Knowledge Management*.
- [10] Karger, D and Stein, L. "Haystack: Per-User Information Environments", February 21, 1997.
- [11] Raskin, J. "The Humane Interface." Addison-Wesley, 2000.
- [12] Berners-Lee, T. *Primer: Getting into RDF & Semantic Web using N3*. <http://www.w3.org/2000/10/swap/Primer.html>.
- [13] *Dublin Core Metadata Initiative*. <http://dublincore.org/>.
- [14] Cutting, D., Karger, D., Pedersen, J., and Tukey, J. "Scatter/gather: A cluster-based approach to browsing large document collections." *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 318-329.