Multiple Open Services: A New Approach to Service Provision in Open Hypermedia Systems

Uffe Kock Wiil, David L. Hicks, Peter J. Nürnberg

Department of Computer Science, Aalborg University Esbjerg
Niels Bohrs Vej 8, 6700 Esbjerg, Denmark
{ukwiil,hicks,pnuern}@cs.aue.auc.dk
phone: +45 7912 {7623,7632,7666} fax: + 45 7545 3643

ABSTRACT

Over the past decade, hypermedia systems have become increasingly open, distributed, and modular. As a direct result of this, open hypermedia systems have been increasingly successful in providing middleware services such as linking to a large set of clients. This paper presents a new approach to service provision in open hypermedia systems based on the concept of multiple open services. The overall idea with multiple open services is to rethink the way in which services are provided to clients. The goal is to split up services into components, each of which provides a general, scalable, and functionally independent (orthogonal) service. This results in a highly flexible architectural framework that can serve as a vehicle to further investigate many of the open issues relating to open hypermedia systems. The approach can be viewed as a natural next step in the evolution towards more open, distributed, and modular hypermedia systems. The concept of multiple open services is described in detail, and a proof of concept implementation called Construct is presented.

KEYWORDS: open service provision, hypermedia, component technology, Construct, middleware services, component-based open hypermedia systems

1 INTRODUCTION

Over the past decade, hypermedia systems have become increasingly open, distributed, and modular [26]. Open hypermedia systems typically provide different middleware services such as hypermedia authoring, hypermedia browsing, collaboration, and distribution. These services are provided through a common interface that allows an open set of desktop applications to make use of the services. Prominent examples of open hypermedia systems developed in the past decade include Microcosm [9], DHM [7], Chimera [2], HOSS [14], and HyperDisco [30]. These systems have been successful in providing open hypermedia middleware services, especially linking, to a wide range of desktop applications (e.g., MS Word, MS Excel, Netscape, MS Internet Explorer, Emacs, and FrameMaker).

Historically, open hypermedia systems provided multiple, often only tangentially related services through a single interface that made it difficult or even impossible to integrate one service without having to integrate all of the services provided by the interface. Consider, for example, HyperDisco, which does not allow the use of its versioning and collaboration services independent of the other services it provides (e.g., its linking services). The collaboration and versioning services are only available as an integral part of some other service. These service dependencies are not natural or necessary, and are a direct consequence of the architecture of HyperDisco.

Research on open hypermedia systems has progressed substantially, especially with the ongoing standardization work in the Open Hypermedia System Working Group (OHSWG) [15]. The OHSWG work has created a focus on the benefits of developing middleware services that have well-defined interfaces and are wrapped in separate components. This has resulted in (de facto) standards for middleware services (e.g., the navigational hypermedia standard OHP-Nav [18]) that enable interoperability between standard-compliant open hypermedia systems as demonstrated at past ACM Hypertext Conferences [5]. However, this process is far from complete; there are still many issues that need to be investigated further. The service decoupling and modularization has so far only been applied to the upper layers of the architecture with some of the services that are directly used by desktop applications such as navigational hypermedia services. The lower layers of the architecture that provide foundation services (such as structure storage, concurrency control, notification control, access control, and version control) have yet to be properly modularized and standardized.

This paper presents a new approach to service provision in open hypermedia systems. The overall idea with multiple open services is to rethink the way in which services are provided at *all* layers in an open hypermedia architecture. The goal is to split up services into components, each of which provides a general, scalable, and functionally independent (orthogonal) service. This results in a highly flexible architectural framework that can serve as a vehicle to investigate further many of the open issues relating to open hypermedia systems. The approach can be viewed as a natural next step in the evolution towards more open, distributed, and modular hypermedia systems, that are based on standards and support interoperability at all layers.

Section 2 outlines the previous work, describes the concepts, and discusses the implications of the multiple open services approach. Section 3 describes an example implementation of a multiple open services environment named Construct, and discusses open issues and future work. Section 4 concludes the paper.

2 MULTIPLE OPEN SERVICES

This section describes the previous work that inspired this new approach, elaborates on the concept of multiple open services, and discusses the implications of the approach.

2.1 How Did We Get Here?

Hypermedia service provision has evolved tremendously since the first monolithic hypermedia systems were developed. Current hypermedia systems are more open and modular with general and extensible system components that provide well-defined services. These services are becoming increasingly standardized within the research community, as the past decade of work has led to a great deal of consensus on the form of at least the most common of them [25]. In the following, three early stages in the evolution of hypermedia systems are identified and described[1].

The earliest hypermedia systems were monolithic (beginning in the 1960s). A monolithic system (e.g., KMS [1] or NoteCards [8]) consists of a closed framework that encompasses all functionality. The monolithic trend continued to influence hypermedia system design for several years. The main problem with monolithic systems is that they do not provide their services to external (third party) applications. The next evolutionary stage in hypermedia system design is client-server hypermedia systems (beginning in the late 1980s). A client-server system (e.g., HyperBase [19], DGS [20], or WWW [3]) contains one or more server processes that provide services to an open set of clients. The main problem with these systems is that the services often enforce a particular data model on their clients in order for them to use the provided services. Some client-server systems also required that clients subscribe to entire sets of functionality (e.g., require that the server handle data storage in addition to the other services it may provide). Thus, applications have to be custom made for the system or must be substantially updated to conform to the server's data model.

Later came work on open hypermedia systems (beginning in the early 1990s). Like a client-server system, an open hypermedia system (e.g., Microcosm [9], DHM [7], or Chimera [2]) provides its services to an open set of clients. However, these services are provided in a more open manner that allows third party applications to make use of them. Applications are not required to subscribe to entire sets of functionality in order to use certain services (e.g., clients can continue to use their "usual" storage service, such as the file system, and do not have to subscribe to a particular data model or data format to use the services).

[1] A more complete description of the evolution of open hypermedia systems can be found in [26].

Most recently, open hypermedia systems are beginning to use component frameworks (e.g., HOSS [14] or HyperDisco [30]), prompting the introduction of the more specific term component-based open hypermedia system (CB-OHS). As part of the transition to component frameworks, services were wrapped in components with well-defined interfaces. Although (CB-)OHSs in many ways represent an improvement over previous work, some problems with service provision still exist. We illustrate these problems using HyperDisco as a case study.

2.2 Case Study: Service Provision in HyperDisco

HyperDisco is an example of an early CB-OHS that provides multiple services. HyperDisco provides different types of services to participating applications:

- *Integration.* This refers to a service that allows integration of existing services (e.g., scripts, programs, and software agents), applications (e.g., editors, browsers, email systems, and spreadsheets), and stores (e.g., databases, file systems, and CD-ROM's) with HyperDisco.

- *Interoperability.* The interoperability service allows HyperDisco to interoperate with other OHSs as well as with integrated applications, stores, and services.

- *Distribution.* The distribution service allows HyperDisco system components (workspaces and tool integrators) and integrated applications, services, and stores to operate in a massively distributed setting such as the Internet. Documents and structure can be retrieved and stored to/from workspaces located anywhere on the Internet.

- *Navigational hypermedia authoring.* This refers to a service that allows hypermedia links to be created between different documents.

- *Navigational hypermedia browsing.* This refers to a service that allows hypermedia links to be traversed.

- *Collaboration.* The collaboration set of services allows users to engage in asynchronous and synchronous collaborative work settings when creating both documents and structure. The set of services includes event notification, locking, and access control services.

- *Versioning.* The versioning service allows documents registered with HyperDisco to be versioned.

HyperDisco defines different levels of integration that a client can engage in [31]:[2]

[2] Davis et al. [6] and Whitehead [24] provide important experiences and models for integration of desktop applications with open hypermedia systems.

- *Launch-only.* This is the most basic level of integration. At this level of integration, HyperDisco can "launch" a client to display the endpoint of a link.

- *Partial integration.* This represents the intermediate levels of integration. This subsumes the entire spectrum of levels from launch only to full.

- *Full integration.* This is the highest level of integration. Clients integrated at this level make use of all the available services.

Obviously, the tighter the integration, the more requirements are placed on the client. The integration service is needed to use any other service. Subscribing to the integration service also means subscribing to the interoperability and distribution services. Once a client subscribes to the integration service, navigational hypermedia authoring and browsing services can be added. Once the hypermedia services have been added, the collaboration and versioning service can be added. Figure 2 gives a conceptual overview of the dependencies between services in HyperDisco.

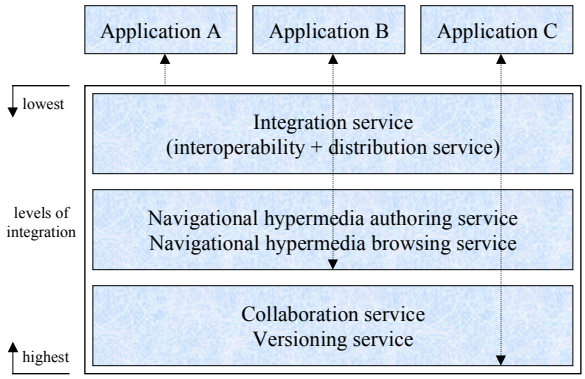

Figure 2: Conceptual overview of dependencies between services provided in HyperDisco. Applications A, B, and C have been integrated at different levels.

In Figure 2, application A is "launch-only"; it can only serve as the destination of a link. When a link is traversed to a document under the control of application A, HyperDisco launches application A to display the document. No hypermedia services are available in application A; it can be considered a hypermedia "dead-end". Application B is integrated at an intermediate level where it makes use of the navigational hypermedia services (both authoring and browsing). Thus, application B can serve as both the source and destination of links; selections inside documents can serve as anchors. Application C is integrated at the highest level (full integration). The collaboration services allow application C to operate in a multi-user environment engaged in both asynchronous and synchronous work settings. The versioning service allows documents under the control of application C to be versioned.

These dependencies between services are not optimal. It is, for instance, not possible to subscribe to HyperDisco's collaboration service without also having to subscribe to the hypermedia services. As discussed below, service provision in HyperDisco is not open as defined in Section 2.3. The orthogonality and scalability requirements are not met.

2.3 Definition

An environment that provides multiple open services has three basic requirements. Firstly, the environment should be based on an open architectural framework. Secondly, the provided services should be generally available. Finally, the provided services should themselves be open. The first two requirements concern the framework (environment) that hosts the services. Most component frameworks (e.g., CORBA [4] or JavaBeans/RMI [10]) already fulfill these two requirements.

Requirement 1. Open architectural framework. An open architecture should consist of an open set of inter-operating components. The set of components is open in the sense that new components can be added and existing ones can be modified. Each component should provide a set of well-defined services through a well-defined interface. An open architecture should be based on a "plug and play" metaphor, allowing, at run-time: insertion of new components with new services; replacement of existing components by new ones; and, removal of existing components.

Requirement 2. General availability. Services should be available on most major computing platforms (such as Sun workstations, PCs and Macintoshes), for most major operating systems (such as Unix, Windows, and MacOS), and to applications written in many different programming languages. Services should be available to an open set of clients in the computing environment. Thus, services should be provided by computing entities or components in the computing environment that are accessible by essentially all applications (i.e., provided by middleware or operating system components).

For purposes of this paper, the most interesting of the requirements is the third one, which concerns the service itself. Thus, the rest of the paper will focus on this requirement.

Requirement 3. Open service. The important requirements of an open service are:

Requirement 3.1. Orthogonality of open services. An open service should be functionally independent of other services used by participating applications (e.g., storage and display services). Applications should be able to use an open service without altering their use of the existing services available in the environment.

Example 1. The integration and use of an open hypermedia linking service by Emacs should not override or change the way that Emacs uses a file system based storage service. This applies to both the availability of the storage service in the user interface (a menu item and the "CTRL-X + CTRL-S" key sequence) and the storage format used (ASCII text).

Thus, the integration should not alter the existing user interface of the storage service (e.g., by placing another service on the "CTRL-X + CTRL-S" key sequence). Likewise, it should not alter the storage format (e.g., by adding anchor and link markup in the text files, as in HTML files).

Example 2. When developing a new structure service, say a metadata service that itself subscribes to one or more foundation services (e.g., structure storage and access control), these foundation services should be functionally independent (orthogonal). Thus, subscription to one foundation service should not in any way overlap or interfere with subscription to other foundation services.

Requirement 3.2. Generality of open services. An open service should be general enough to be useful across applications. The service should be operational both internally in the application and across to other applications of the same type or other types such as, for example, "cut, copy, and paste" services.

Requirement 3.3. Scalability of open services. An open service should provide different levels of its services. Applications need not integrate advanced levels of a service if only a basic level is desired.

It is important to notice that the three open service requirements should apply to all layers in an open hypermedia system as indicated with the two orthogonality examples above.

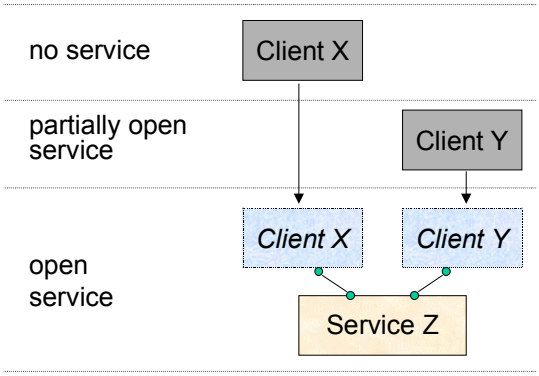

Figure 1: The concept of open service provision.

Figure 1 illustrates, in an abstract and generic manner, the concept of open service provision. Two different types of clients (e.g., desktop applications or other services), client X and client Y, exist in the example computing environment. The environment also contains a service component that makes a specific service (called the Z service) available to an open set of clients. The darker shaded boxes in the upper part of Figure 1 illustrate the status of both clients before their integration of the facilities provided by the Z service. Client X does not provide the Z service at all. Client Y does provide the Z service, but only in a partially open manner, since the Z service can only be used within the same instance of client Y, and not across to other instances of the client. As indicated by the arrows in Figure 1, both clients integrate the functionality of the Z service component to augment the functionality they provide. Client X uses the service component to add the Z service to the overall set of services it provides without altering or overriding any existing service already available in the client. Client Y uses the Z service component to extend the services it provides and allow Z service functionality to be used in an open way across to other instances of the Y client, or other types of clients (e.g., instances of client X).

2.4 What Are the Implications?

Natural and very relevant questions to ask, when confronted with a new approach like the multiple open services one, are "How does this improve earlier work?" and "What are the drawbacks?" This section will answer the first question. An answer to the second question will be given in Section 3.5, which elaborates on open issues and future work.

The multiple open service idea is to decompose services into smaller chunks that can be used independently of one another. Many applications may still have to subscribe to a wrapper service (similar to the integration service in HyperDisco) in order to be able to subscribe to another service. However, applications should be able to subscribe to any service from the overall set of services. Figure 3 gives a conceptual overview of the multiple open services idea.

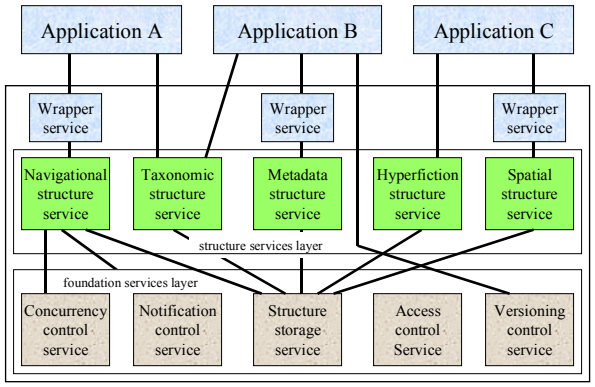

Figure 3: Conceptual overview of open service provision. Applications A, B, and C are clients of different combinations of services. Likewise, the five structure services are clients of different combinations of foundation services. The combination of services shown here are arbitrary and may not make any sense.

The important things to notice in Figure 3 in connection with the multiple open services approach are:

1. Services belong to a layer in the architecture depending on the type of service they provide (e.g., application, structure service, or foundation service layer).

2. All services at a layer (e.g., foundation layer) must comply with the open service requirements, hence they must be functionally independent, general and scalable.

The open service requirements can only be enforced among services in the same layer. Many structure services may subscribe to the same foundation service. For example, both a navigational and a taxonomic structure service may provide access control as an integrated part of their structure service. Thus, these two structure services cannot be functionally independent (orthogonal) with respect to access control.

3. Each layer is open to new services as long as the requirements for open services are ensured. Thus, it is possible to add new types of structure services (e.g., argumentation support [21] and workflow services [22]) to the set of existing structure services in Figure 3.

4. Any component can in principle be a client of any other component. In Figure 3, application B is a client of wrapper, structure, and foundation services. However, in most cases services at one layer will only be clients of services at the layer immediately below.

The rationale behind the multiple open services approach and more specifically the quest for orthogonality, generality, and scalability is to develop a highly modular and flexible architectural framework that can serve as a vehicle to investigate further many of the open issues relating to open hypermedia systems. We believe that the multiple open services approach has the following advantages:

- *Standardization and interoperability*. It is a natural next step in the standardization process that allows the OHSWG to broaden its standardization and interoperability effort to cover all layers of open hypermedia systems and not just the interfaces between applications and structure services. If all interfaces in an open hypermedia system are standardized, interoperability will be achieved at all layers in open hypermedia systems. This would be a huge benefit for the open hypermedia research community and for users of open hypermedia systems.

- *Service modularization*. Decoupling of services is a distinct advantage for performing research on open hypermedia systems. By modularizing the services, the interactions between services can be more easily explored. Modularized services also allow each service to be considered independently, and for each service or class of services to be an independent focus of attention and research. This is a step in the right direction towards developing a deep understanding of the universe of open hypermedia services.

Besides these two overall advantages, we also believe that the multiple open services approach will turn out to be useful for developers of services and application integrators.

- *Application integrators* will benefit from having a set of well-defined services that they can integrate into their application on a piecemeal basis. We believe that by making the services modular and functionally independent, they become easier to comprehend by application integrators. Section 3.4 provides an example of multiple open services integration in Emacs.

- *Service developers* will only have to develop a service once. In a system with service dependencies, it may be necessary to provide the same service in different incarnations. For example, the collaboration service in HyperDisco would need to be replicated and retrofitted if a new type of structuring service was introduced (e.g., spatial hypermedia [11], taxonomic hypermedia [16], or argumentation support [21]).

3 CONSTRUCT: AN EXAMPLE IMPLEMENTATION

This section provides a brief overview of the Construct environment, describes two example application integrations, explains how the Construct navigational structure service meets the requirements for open service provision, and provides an example of managing multiple open services in an application. The section concludes with a discussion of open issues and future work raised by the multiple open services approach.

3.1 System Overview

Construct is a component-based open hypermedia system [26] developed at Aalborg University Esbjerg and Aarhus University (both in Denmark). The Construct environment [27] (Figure 4) consists of different categories of software components[3]:

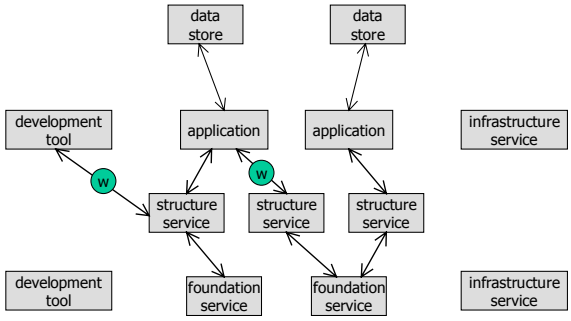

Figure 4: An overview of the Construct environment. The circles containing a "w" indicate a wrapper service.

- *Applications*. This category includes desktop applications (e.g., Netscape, MS Word, and Emacs) that have been integrated (modified, extended, or wrapped) to be able to make use of the Construct structure services.

- *Wrapper services.* A wrapper is a service that allows a legacy application (and middleware services, data stores, etc.) to be integrated with the Construct environment. Natively compliant applications do not need wrappers – all other applications do.

[3] An earlier version of Construct that predates the development tools is described in [26].

- *Structure services*. Integrated applications can make use of different types of structure services to organize data located in data stores. Each type of structure service provides a different set of structural abstractions (e.g., navigational, spatial, taxonomic, argumentation, workflow, and collaboration) that supports different application domains.

- *Foundation services.* Foundation services provide the very basic services in the environment that most other services depend on. Example foundation services are:

  - *Structure stores*. A structure store is a software component that manages storage and retrieval of structure. Structure stores can handle different types of structural abstractions.

  - *Multiuser and collaboration services.* This category includes services such as concurrency control, notification control, and access control.

  - *Versioning services.* Services that allows structural abstractions to be versioned.

- *Data stores*. A data store could be a file system, a database, a CD-ROM, etc.

- *Infrastructure services*. This category includes general services that enable the individual components of the environment to co-exist. Examples of these services are service discovery, naming, and location.

- *Development tools*. The development tools assist in the development of new services in the environment. Currently, three development tools exist:

  - *UML Tool*. This tool allows the developer to specify new services by drawing UML diagrams in a graphical user interface. UML Tool translates the UML diagrams into IDL specifications.

  - *Emacs.* This well-known editing tool is used to create IDL specifications and Java code as well as documentation of various kinds. Emacs is considered a development tool of the Construct environment due to its integration with the navigational structure service.

  - *Construct Service Compiler (CSC)*. The CSC takes IDL specifications as input and generates service skeletons in Java. Service skeletons are wrapped in components that can automatically run as part of the Construct environment. The developer must add some code to the skeletons to define the semantics of the generated operations.

  Development tools can also be clients of structure services. For example, UML Tool and Emacs are clients of the navigational structure service (see Sections 3.2, 3.3, and 3.4). Thus, it is possible to link from classes or fields in the UML diagrams to, for example, design rational located in a text file under the control of Emacs.

The Construct environment is based on well-known technologies for building component frameworks including JavaBeans and RMI. The entire system is coded in Java.

3.2 Example Application Integrations

UML Tool. UML Tool was integrated with the Construct navigational structure service using a wrapper written in Java. The wrapper communicates with UML Tool using a simple text based protocol on top of TCP/IP. The wrapper translates the requests from UML Tool into Construct navigational operations and sends them to the navigational structure service. Responses from the navigational structure service are translated back to the text-based protocol and sent to UML Tool.

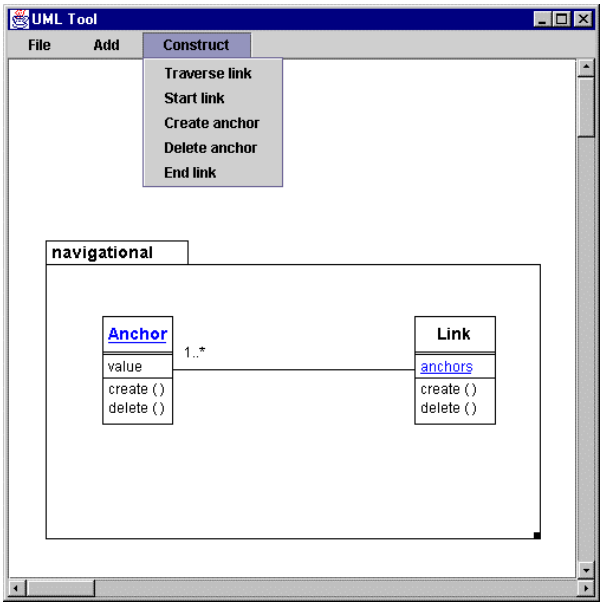

Figure 5: UML Tool as a Construct application. Anchors are displayed on screen as blue, underlined text.

UML Tool itself has been extended with three kinds of Java methods: those that provide the user interface to the navigational hypermedia services, those that handle responses from the wrapper and those that handle communication to the wrapper. Figure 5 shows a screen dump of UML Tool when running as an application of the Construct navigational structure service. A new menu (named Construct) with five menu items (Traverse link, Start link, Create anchor, Delete anchor and End link) is added to the interface. This is the only visible indication that UML Tool has been extended with navigational hypermedia services.

Vital. Vital [21] is a hypermedia-based collaborative learning tool developed at GMD-IPSI using the Coast toolkit [17]. Vital is developed in Visual Works Smalltalk, which runs on both the Sun Sparc and the MS Windows platforms. Vital is integrated in much the same way as UML Tool. A wrapper

written in Java performs transformation operations similar to those of the UML Tool wrapper. A Smalltalk code library extends Vital with a menu and the necessary communication functions. Figure 6 shows a screen dump of Vital when running as an application of the Construct navigational structure service. A new window with the available navigational hypermedia functions (Connect, Disconnect, Create Anchor and End Linking) is added to Vital's interface (see the small window on the left side of Figure 6). Vital links span pieces of information internally in Vital, while Construct links can connect Vital items to pieces of information outside of Vital (e.g., to some words in a text file displayed by Emacs or to a part of a UML diagram displayed by UML Tool).

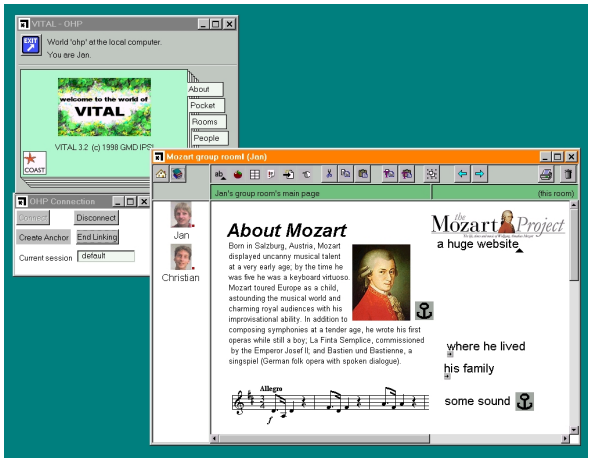

Figure 6: Vital as a Construct application. Vital anchors are shown as arrow icons and Construct anchors are shown as anchor icons attached to the linked pieces of information.

3.3 Open Service Provision

This section will make the case that the navigational hypermedia service is in fact an open service as defined in Section 2.1. Section 3.5 discusses some of the issues raised by having multiple open services running along side each other in the same environment.

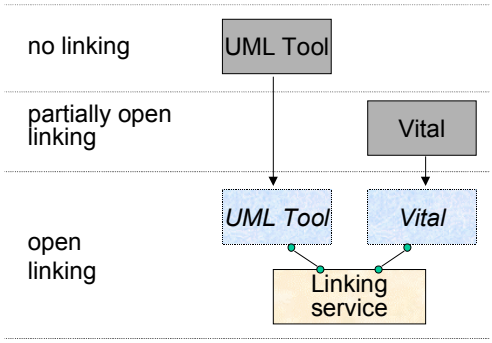

Figure 7: Open linking provision in UML tool and Vital.

The starting points for the integrations of UML and Vital were very different (Figure 7). UML Tool did not provide general navigational hypermedia services (linking). Vital provided a partially open form of linking. Users could create their own links, but they could only associate objects that resided inside (under the control of) Vital.

In both cases, the result of the integration (Figure 7) is a new set of operations that allows users to create links that can associate objects and files under the control of different applications. Thus, the navigational hypermedia service is general enough to be useful across different applications (generality requirement). It is interesting to note that the augmented Vital has two types of linking services that co-exist, the original "internal" (partially open) linking service and the new open linking service. Thus, the navigational hypermedia service is utilized at two different levels by UML Tool and Vital (scalability requirement). The existing services of Vital and UML Tool remain untouched by the navigational hypermedia augmentation. In other words, the navigational hypermedia service is orthogonal to any existing service in Vital and UML Tool (orthogonality requirement).

3.4 Managing Multiple Open Services in Applications

This section presents an example of management of the multiple open services in an existing desktop application. Emacs subscribes to both the Construct metadata service and navigational service. The example serves as a first proof of concept of the multiple open services approach. The example consists of a set of screen shots (with explanatory text) that step by step show (and explain) how multiple open services are managed in Emacs.

Emacs is started with the Construct extension and a menu named *Construct* appears in Emacs' interface. A file called *navigational.txt* is loaded into a buffer in Emacs.

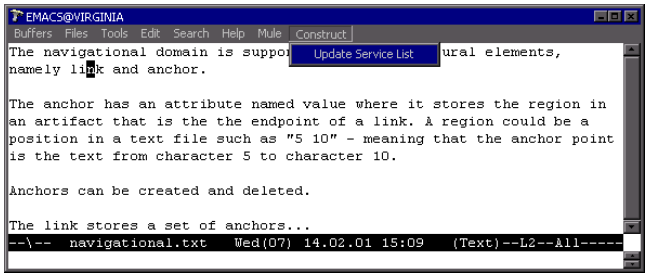

Figure 8: The initial Construct menu in Emacs before the list of services is updated.

The *Update Service List* command in the *Construct* menu is selected (Figure 8). Emacs contacts the Construct *Service Information Manager* (SIM) to request a list of available open services. Based on the response from the SIM, Emacs updates the *Construct* menu to display the set of available services (*Metadata Service* and *Navigational Service*) – see Figure 9. The *Update Service List* command continues to be available and operational throughout the session with Construct. Hence, the set of available Construct services in Emacs is dynamically reconfigurable. The *Load Metadata Service* command is activated and the *Metadata* menu appears in the interface (Figure 10).

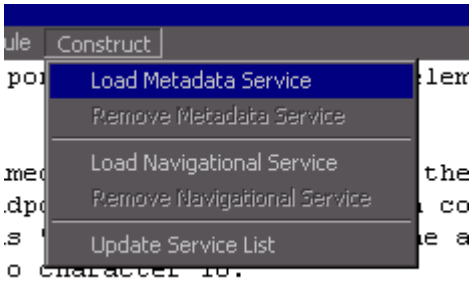

Figure 9: A close-up view of the Construct menu after updating the list of services.

The *Load Navigational Service* command is also activated (Figure 10) and the *Navigational* menu appears in the interface (Figure 11).

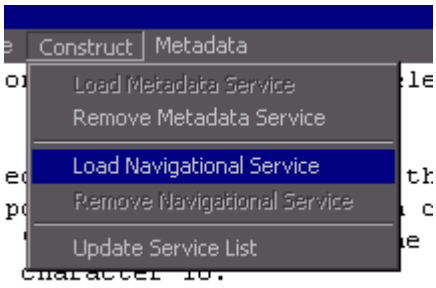

Figure 10: A close-up view of the Construct menu after loading the Metadata Service

A service can be removed again from the interface by activating the remove command (i.e., *Remove Navigational Service* will remove the *Navigational* menu from the interface). A removed service can later be loaded again.

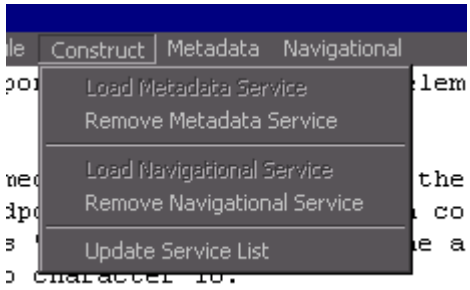

Figure 11: A close-up view of the Construct menu after loading both services.

Now the navigational service is used to create links from selected portions of text in *navigational.txt* to other linkable data sources (e.g., UML diagrams inside UML Tool) – see Figure 12. Next, the metadata service is used to create a metadata record for *navigational.txt* (Figure 13). The record consists of three keys (Date, Keywords, and Author) and their corresponding values.

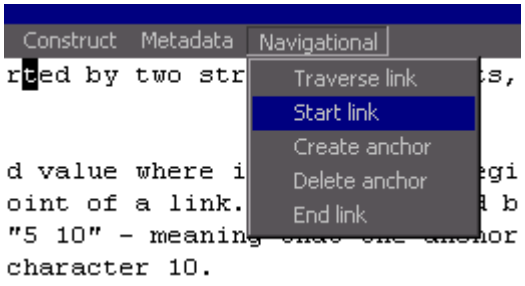

Figure 12: A close-up view of the Navigational menu.

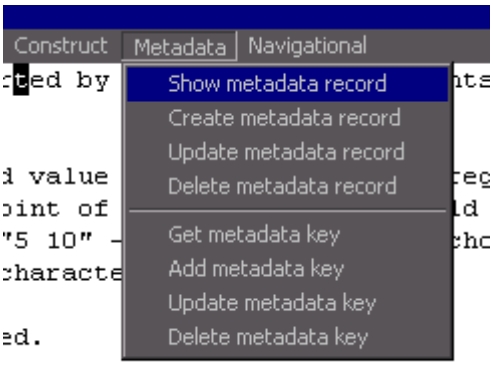

Figure 13: A close-up view of the Metadata menu.

The result of the metadata and linking operations can be seen in the last screen shot (Figure 14) that shows how the two Construct services co-exist in Emacs.

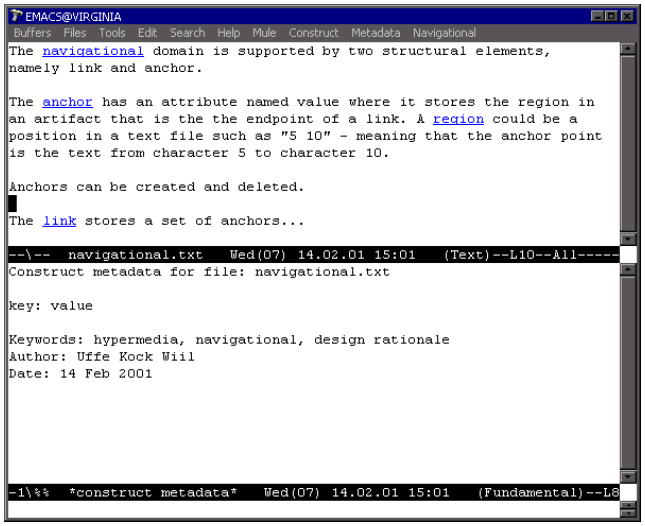

Figure 14: Emacs using the Construct Navigational services and Metadata services at the same file on the same time.

The buffer in the top part of the screen displays *navigational.txt* overlaid with navigational structure. Anchors are displayed on screen as blue, underlined text. The buffer in the lower part of the screen displays the metadata record for *navigational.txt*.

3.5 Open Issues and Future Work

The fact the services are decoupled and modularized at all architectural layers raises a number of issues regarding the interoperability of the decomposed services.

The standardization work of the OHSWG is addressing the issues of interoperability at the application layer [25]. Interoperability at this layer, allows clients of one open hypermedia system to work with other open hypermedia systems [18]. Nürnberg et al. first discussed the issues of interoperability among structure services, specifically the issues involved in having one structure service interpret the structure of another structure service (e.g., a navigational structure service that can interpret spatial hypermedia structures) [13]. Later, Millard et al. introduced the FOHM model, which addresses interoperability between three particular structure services (spatial, navigational, and taxonomic) by defining a common data model for these three structural domains [12].

The multiple open services work raises an additional set of issues relating to interoperability. It is now relevant to discuss what interoperability means at all layers in an open hypermedia system.

Considering interoperability between different foundation services is interesting. This implies the need for each foundation service to be able to interpret (translate) the abstractions manipulated by other foundation services. Is this a realistic or even useful goal? For example, consider an access control service that can interpret the abstractions of a concurrency control service. This particular example may make sense, since the fact that a document is locked could be interpreted as a special level of access control denying other people the ability to open the document. Clearly, we cannot yet say that translating from any given foundation service to any other is a useful undertaking, but it does seem that at least some of these translations may be useful.

One way to discuss interoperability is to distinguish between horizontal and vertical interoperability. Horizontal (intra-layer) interoperability covers the cases where a service at one layer can interpret (translate) abstractions provided by other services at the same layer. Horizontal interoperability has been the focus of attention of the work by the OHSWG (application layer) and of the work by Nürnberg et al. and Millard et al. (structure service layer). However, there are still many unsolved issues relating to horizontal interoperability [13]. Vertical (inter-layer) interoperability covers the cases where a service at one layer can interpret and use abstractions provided by services at other layers. Vertical interoperability occurs in layered systems that provide different abstractions at different layers. This type of interoperability has been investigated and accomplished in many open hypermedia systems. Construct, for example, provides a basic structural unit in its structure store (foundation layer). All structure services use this unit to provide more advanced structural abstractions to their clients (e.g., contexts, links, and anchors).

Another issue raised by the multiple open services approach is that of having layers of services within layers. This is best illustrated with an example at the foundation layer. Consider a new service that combines some of the existing foundation services (e.g., structure storage and version control) to provide a versioned structural unit. This type of service would still conceptually belong to the foundation layer due to the services that it provides. What does this mean? Can such new combination services be considered open services according to the definition in Section 2.1?

We plan to investigate further these open issues in our future work with Construct both at a conceptual level and at a practical level involving development of additional services and integration of additional applications.

4 CONCLUSIONS

The multiple open services approach is a natural continuation of the ongoing trend to make open hypermedia systems more open, distributed, and modular and provide a wider variety of services. This paper defines a set of desirable characteristics for decoupling and modularizing services. Developers of open services are charged with the tasks of designing the right kinds of services (generality requirement), designing the right levels of services (scalability requirement), and deciding on the right grouping of services (orthogonality requirement).

The primary advantage of this new approach to service provision in open hypermedia systems is a highly flexible and modular architecture that has benefits for future research on open hypermedia systems. Each service can be developed and investigated independently. The OHSWG standardization and interoperability effort can migrate into all layers of open hypermedia systems. At the same time, decoupling and modularization of services opens up a new set of issues relating to interoperability that needs to be addressed in future conceptual and practical work.

ACKNOWLEDGMENTS

We wish to thank Jörg Haake for funding a research trip to GMD-IPSI, Darmstadt in July 1998, where some of the preliminary ideas behind multiple open services (later presented in [28] and [29]) were discussed and addressed in early prototype implementations. The Vital integration with the Construct navigational service is one visible result of this collaboration.

REFERENCES

1. Akscyn, R., McCracken D., and Yoder, E. 1988. KMS: A distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM, 31*(7), 820-835.
2. Anderson, K., Taylor, R., and Whitehead, E. J. 1994. Chimera: Hypertext for heterogeneous software environments. In *Proceedings of the 1994 ACM European Conference on Hypertext*, (Edinburgh, Scotland, Sep), ACM Press, 94-107.
3. Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secret, A. 1994. The World-Wide Web. *Communications of the ACM, 37*(8), 76-82.
4. CORBA. http://www.corba.com

5. Davis, H. C., Millard, D. E., Reich, S., Bouvin, N. O., Grønbæk, K., Nürnberg, P. J., Sloth, L., Wiil, U. K., and Anderson, K. M. 1999. Interoperability between hypermedia systems: The standardization work of the OHSWG. In *Proceedings of the 1999 ACM Hypertext Conference*, (Darmstadt, Germany, Feb), ACM Press, 201-202.
6. Davis, H. C., Knight, S., and Hall, W. 1994. Light hypermedia link services: A study of third party application integration. In *Proceedings of the 1994 ACM European Hypertext Conference*, (Edinburgh, Scotland, Sep), ACM Press, 41-50.
7. Grønbæk, K., and Trigg, R. 1999. *From Web to Workplace – Designing Open Hypermedia Systems.* MIT Press.
8. Halasz, F., Moran, T., and Trigg, R. 1987. NoteCards in a nutshell. In *Proceedings of the 1987 ACM CHI Conference*, (Toronto, Canada, Apr), ACM Press, 45-52.
9. Hall, W., Davis, H., and Hutchings, G. 1996. *Rethinking Hypermedia – The Microcosm Approach.* Kluwer Academic Publishers.
10. Java Technology. http://www.javasoft.com
11. Marshall, C., and Shipman, F. 1995. Spatial hypertext: Designing for change. *Communications of the ACM, 38*(8), 88-97.
12. Millard, D. E., Moreau, L., Davis, H. C., and Reich, S. 2000. FOHM: A fundamental open hypertext model for investigating interoperability between hypertext domains. In *Proceedings of the 2000 ACM Hypertext Conference*, (San Antonio, TX, Jun), ACM Press, 93 – 102.
13. Nürnberg, P. J., Leggett, J. J., and Wiil, U. K. 1998. An agenda for open hypermedia research. In *Proceedings of the 1998 ACM Hypertext Conference*, (Pittsburgh, PA, Jun), ACM Press, 198-206.
14. Nürnberg, P. J., Leggett, J. J., Schneider, E., and Schnase, J. 1996. HOSS: A new paradigm for computing. In *Proceedings of the 1996 ACM Hypertext Conference*, (Washington, DC, Mar), ACM Press, 194-202.
15. Open Hypermedia System Working Group (OHSWG). http://www.ohswg.org
16. Parunak, H. 1991. Don't link me in: Set based hypermedia for taxonomic reasoning. In *Proceedings of the 1991 ACM Hypertext Conference*, (San Antonio, TX, Dec), ACM Press, 233-242.
17. Schuckmann, C., Schümmer, J., Kirchner, L., and Haake, J. 1996. Designing object-oriented synchronous groupware with COAST. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, (Boston, MA, Nov.), ACM Press, 30-38.
18. Reich, S., Wiil, U. K., Nürnberg, P. J., Davis, H. C., Grønbæk, K., Anderson, K. M., Millard, D. E., and Haake, J. M. 1999. Addressing interoperability in open hypermedia: The design of the open hypermedia protocol. Special issue on open hypermedia, *The New Review of Hypermedia and Multimedia*, 5, 207-248.
19. Schütt, H. A., and Streitz, N. A. 1990. Hyperbase: A hypermedia engine based on a relational database management system. In *Proceedings of the 1990 European Conference on Hypertext*, (Versailles, France, Nov), Cambridge University Press, 95-108.
20. Shackelford, D. E., Smith, J. B., and Smith, F. D. 1993. The architecture and implementation of a distributed hypermedia storage system. In *Proceedings of the 1993 ACM Hypertext Conference*, (Seattle, WA, Nov), ACM Press, 1-13.
21. Smolensky, P., Bell, B., Fox, B., King, R., and Lewis, C. 1987. Constraint-based hypertext for argumentation. In *Proceedings of the 1987 ACM Hypertext Conference*, (Chapel Hill, NC, Nov.), ACM Press, 215-245.
22. Wang, W., and Haake, J. M. 1999. Supporting workflow using the open hypermedia approach. In *Proceedings of the First Workshop on Structural Computing*, Technical Report AUE-CS-99-04, Aalborg University Esbjerg, Denmark, 12-17.
23. Wessner, M., Beck-Wilson, J., and Pfister, H. H. 1998. CLear – a cooperative distributed learning environment. In *Proceedings of the 1998 ED-MEDIA/ED-TELECOM Conference*, (Freiburg, Germany, Jun.), 1876-1877.
24. Whitehead, E. J. 1997. An architectural model for application integration in open hypermedia environments. In *Proceedings of the 1997 ACM Hypertext Conference*, (Southampton, UK, Apr), ACM Press, 1-12.
25. Wiil, U. K. 1997. Open hypermedia: Systems, interoperability and standards. Special issue on open hypermedia, *Journal of Digital Information, 1*(2), (Dec).
26. Wiil, U. K., and Nürnberg, P. J. 1999. Evolving hypermedia middleware services: Lessons and observations. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, (San Antonio, TX, Feb), ACM Press, 427-436.
27. Wiil, U. K., Nürnberg, P. J., Hicks, D. L., and Reich, S. 2000. A development environment for building component-based open hypermedia systems. In *Proceedings of the 2000 ACM Hypertext Conference*, (San Antonio, TX, Jun.), ACM Press, 266-267.
28. Wiil, U. K., Nürnberg, P. J., and Leggett, J. J. 1999. Hypermedia research directions: An infrastructure perspective. Hypertext and Hypermedia Symposium, *ACM Computing Surveys*, 31(4), (Dec).
29. Wiil, U. K. 1999. Multiple open services in a structural computing environment. In *Proceedings of the 1st Workshop on Structural Computing*, Technical Report CS-99-04, Aalborg University Esbjerg, 34-39.
30. Wiil, U. K., and Leggett, J. J. 1997. Workspaces: The HyperDisco approach to Internet distribution. In *Proceedings of the 1997 ACM Hypertext Conference*, (Southampton, UK, Apr), ACM Press, 13-23.
31. Wiil, U. K., and Leggett, J. J. 1996. The HyperDisco approach to open hypermedia systems. In *Proceedings of the 1996 ACM Hypertext Conference*, (Washington, DC, Mar), ACM Press, 140-148.