Repository Support for the Virtual Software Enterprise

John Noll and Walt Scacchi
Information and Operations Management Dept.
University of Southern California
Los Angeles, CA 90089-1421

Abstract

Software development in the future will conducted by "virtual enterprises," consisting of loosely-coupled, widely distributed, autonomous development teams.

Due to geographic separation and the loose cooperative structure of a virtual enterprise, each team will be highly autonomous, managing its own computing environment and tools. Thus, there will be no shared file-system or database to serve as an integrating resource. Similarly, we cannot assume that all teams will agree to adopt and use any one team's object storage manager or data model. Nonetheless, it will be necessary for each team to be able to modify jointly developed documents or process models.

This papers presents a hypertext-based solution called "DHT" that supports software engineering data modeling and management, provides transparent access to heterogeneous, autonomous legacy repositories, and enables an implementation strategy with low cost and effort. In addition, we show how DHT solves the practical problems of sharing data in a virtual enterprise, integrating existing tools and environments, and enacting software processes.

1 Introduction

Contemporary software development environments follow the architectural model shown in Figure 1: software engineers employ a set of development tools to create and evolve software artifacts[1] that are stored in a central repository, which serves as the medium for sharing artifacts among engineers and programmers.

Software development in the future will likely take place in an environment that is much different from this model. Software development teams will be highly decentralized, both physically and organizationally. Software will be produced by loosely coupled, "virtual enterprises," composed of developers from different companies who cooperate on specific projects, then disband to form new alliances for other projects. Participants in these virtual enterprises will retain a high degree of autonomy over their own activities, environments, and data. Nevertheless, they will need to share artifacts with other members, such as requirements, project plans, source code, test results, deliverables, process models, and the relationships among them.

Figure 2 depicts an example of a "typical" future development scenario. A loose collaboration among a customer, consultant, vendor, and software contractor is formed to enhance a legacy system. Each has its own internal tools and data, and each will require access to at least part of the others' data. Additionally, each will need to update and

[1] The term "software artifact" refers to the documents, programs, executables, etc. that form the results of the software engineering process

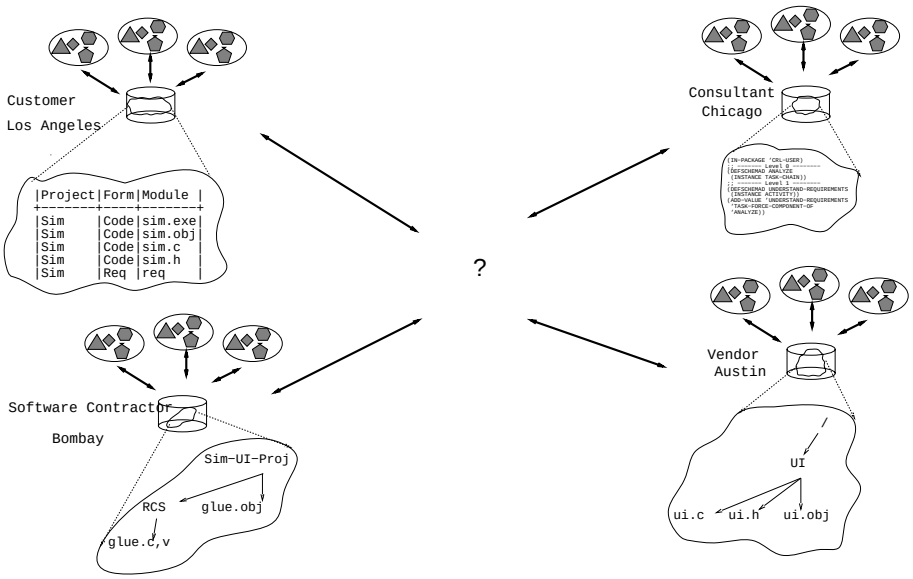

Figure 2: A virtual software enterprise.

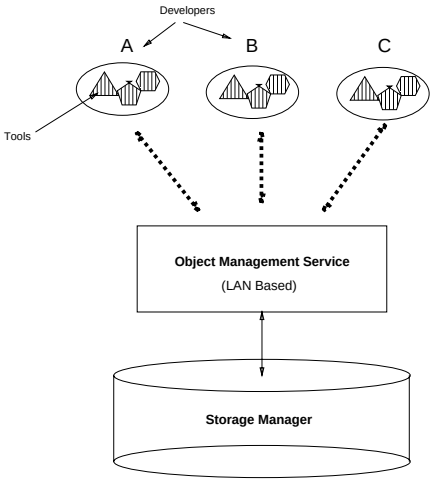

Figure 1: Software development environment architecture.

expand the set of artifacts that represents the project's output and deliverables.

Due to geographic separation and the loose cooperative structure, each team will be highly autonomous, managing its own computing environment and tools. Thus, there will be no shared file-system or database to serve as an integrating resource. Similarly, we cannot assume that all teams will agree to adopt and use any one team's object storage manager or data model. Nonetheless, it will be necessary for each team to be able to modify jointly developed documents or process models.

The virtual enterprise produces a set of shared artifacts, but does not have a central repository where they can be stored. Without a shared database, how can participants retrieve and modify shared objects? This is the problem addressed by this paper: *how to enable sharing of software artifacts among autonomous, distributed development teams, given the absence of a shared repository.*

A solution must meet the following goals:

1. Model software artifacts, versions, and other relationships.

2. Provide transparent access to artifacts and relationships in heterogeneous, autonomous legacy repositories.

3. Preserve the local autonomy of integrated repositories.

4. Integrate existing tools and environments.

5. Support software process modeling and enactment in widely distributed environments.

6. Enable a simple implementation strategy. This goal stems from the purpose of virtual enterprises as a way to react to rapidly changing marketplaces. The cost of a solution's implementation should not defeat this purpose.

We present an approach to software artifact sharing, called the Distributed Hypertext system (DHT), that employs a *hypertext* data model as the common data model for the integration layer. Hypertext is an information management concept that organizes data into content objects called *nodes*, containing text, graphics, binary data, or possibly sound or video, that are connected by *links* which establish relationships between nodes or *sub-parts* of nodes. The resulting graph, called a hypertext *corpus*, forms a semantic net-like structure that can capture rich data organization concepts while at the same time providing intuitive user interaction via browsing.

DHT provides logical integration by representing artifacts using a hypertext data model that augments the traditional node and link model with aggregate constructs (called *contexts*) that represent subgraphs of the hypertext, and dynamic links that allow the global hypertext to evolve automatically as artifacts are created and modified. The data model (described in Section 2) defines the structure of objects in the global hypertext, and the operations (including updates) that may be performed on them.

DHT achieves physical integration with a client-server architecture (also described in Section 2) that provides transparent access to heterogeneous repositories through intermediary components called *Transformers*. Clients are software tools that developers use to access objects in the repositories.

In previous work [15, 16] we have discussed how DHT addresses the first three goals. In this paper, we will focus on tool and environment integration, and software process modeling and enactment within the same straightforward integration and implementation strategy.

This paper is organized as follows: in the next section we present an overview of the DHT architecture and data model. Following, we discuss the DHT approach to tool integration. Then, we present an approach to software process modeling and enactment using DHT-based hypertext browsing. We conclude with a discussion of related research, and our contributions.

2 DHT Architecture and Data Model

This section presents a brief overview of the DHT architecture and data model; a more detailed discussion can be found in [14].

Architecture. The DHT architecture is based on a client-server model. Clients implement all application functionality that is not directly involved in storage management. Thus, a client is typically an individual tool, but may be a complete environment.

Software artifacts are *exported* from their native repository through server components called *transformers*. A transformer exports local objects (artifacts and relationships) as DHT nodes and links, and translates DHT messages into local operations

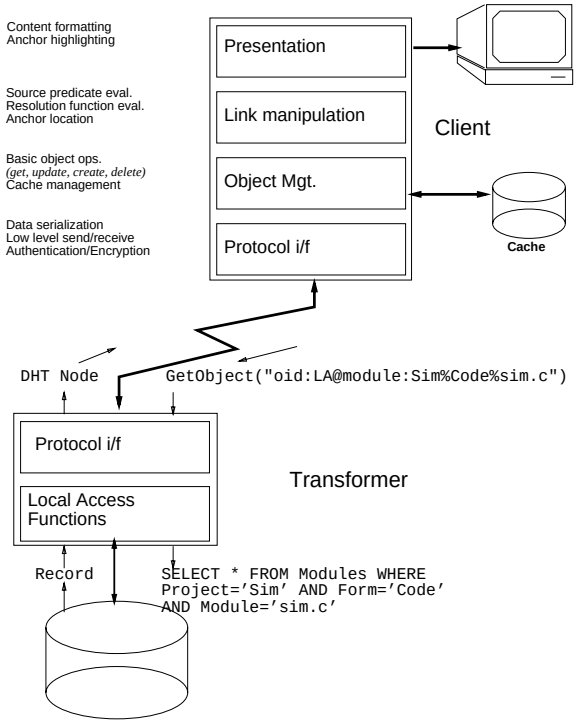

Figure 3: The DHT Architecture.

(see Figure 3). Note that from the repository viewpoint, the transformer appears to be just another application.

A request-response style communication protocol implements the operations specified in the DHT data model (see below), and includes provisions for locating servers and authenticating and encrypting messages. The protocol also provides a form of time-stamp concurrency control [12, 16] to prevent "lost updates".

Data Model. The DHT Data Model consists of four basic objects: *nodes*, that represent content objects such as modules; *links* that model relationships among nodes; *anchors*, that specify the points within node contents that anchor the endpoints of a link; and *contexts*, that enumerate sets of links to allow specification of object compositions as sub-graphs. Nodes, links, and contexts have types, attributes, and unique object identifiers (oids).

A fixed set of operations can be applied to DHT objects: *create*, *delete*, *read*, and *update* an object. A given repository can elect to provide any subset of these operations, as appropriate for the level of access it intends to provide. In addition, any operation can be performed by a single repository on its own objects; cooperation among repositories is not required.

3 Tool Integration

In practice, clients are software engineering tools and environments, most of which will exist before integration by DHT. It is impractical to expect users and organizations to discard their favorite tools in favor of new tools that understand DHT. Therefore, DHT includes a strategy for migrating existing tools to the DHT environment, and a cache mechanism to ensure that the performance of migrated tools, when accessing remote nodes and links, approaches that of local object access.

The migration strategy specifies four levels of integration:

Level 0 At level 0, tools are not integrated at all. They exist unmodified, and require auxiliary tools to interact with DHT on their behalf. Auxiliary tools simply perform node retrieval and update, and link resolution, to and from a tool's standard input/output, or files in the local file-system.

Level 1 Level 1 integration treats DHT nodes as file-like objects. Tools use file-system calls like *open()*, *read()*, *write()*, etc. to access a node's contents, passing a string representation of the node's oid rather than a file pathname. Level 1 integration can be accomplished without recompiling or modifying source code; simply relinking the tool with a DHT compatibility library (see Section 3) is all that is required. Note, however, that Level 1 tools do not have access to links.

System call	Equivalent DHT operation
open()	DhtRead()
access()	same as open()
read()	read() from contents file
write()	write() to contents file
close()	DhtUpdate(); DhtSync()
stat()	stat() on contents file

Figure 4: DHT file-system emulation functions.

Level 2 At level 2, a tool is aware of links as relationships among objects, and can follow them. This awareness does not appear at the user interface.

An example of a level 2 tool is a document compiler, that resolves links of type "Include" to incorporate text from other nodes into a source node.

Level 3 At level 3, a tool integrates hypertext browsing and linking into its user-interface, which may require extensive modification to the tool's source code. Fortunately, many tools incorporate extension languages or escapes to external programs that can be used to implement linking without re-compilation; this technique was used to implement the DHT editor using GNU Emacs Lisp.

File system emulation. A vast legacy of tools uses the file-system as its repository. These applications read and write objects as files through the file-system interface, typically by calling "standard IO" [18] library routines supplied for the application's implementation language. Our goal to provide a reasonable-cost implementation strategy, precludes requiring that all of these tools be modified to use the DHT application interface in place of the file-system library.

To solve this problem, the DHT architecture exploits the file-like nature of DHT atomic nodes to provide a file-system emulation library. This library intercepts the Unix file-system calls and converts them to DHT access operations when strings encoding DHT object identifiers are passed as the pathname argument. The entry points are shown in Figure 4.

To enable a tool for DHT access, one simply re-links it with the emulation library. Thus, the tool will continue to function as before when invoked with real file names, yet will retrieve contents from the DHT object cache (described below) when DHT object identifiers are used.

Object Caching. Many DHT objects change slowly; others see frequent access during a short period of time. To improve access latency and reduce transformer loads, it is desirable to cache frequently used objects that may be from repositories accessed over the Internet.

A cache layer is built into the the basic request interface to provide transparent node and link caching. The cache is maintained in the local file-system; node contents are cached in separate files to support the file-io emulation library discussed above, while links and node attributes are cached in a hash table. Clients call a set of object access functions to retrieve objects through the cache layer; these are listed in Figure 5.

Each DHT object has a "time-to-live" attribute that specifies the length of time an object in the cache is expected to be valid. The cache layer uses this attribute, set by the transformer when the object is retrieved, to invalidate objects in the cache upon access. An administrative function exists to sweep through the entire cache and remove all objects whose time-to-live has expired.

The time-to-live attribute is not a guarantee of validity, however. Certain shared objects may be updated frequently by multiple clients. To allow such clients to verify that requested objects have not been modified by another client, the cache layer can be configured with four cache policies to sup-

Function	Description
DhtRead()	retrieve the specified objects.
DhtReadContents()	return the file containing the object's contents.
DhtUpdate()	update the cached copy of an object.
DhtSync()	update the specified object at the transformer.
DhtSource()	evaluate a link's source predicate on a node.
DhtResolve()	resolve a link given a source node and anchor.

Figure 5: Cache layer interface.

port specific application needs:

1. Never use the cached copy; always retrieve an object from the repository.
2. Use the cached copy if its time-to-live has not expired.
3. Use the cached copy if it has not been modified; verify this by retrieving the object's timestamp from the repository.
4. Always use the cached copy if present, regardless of its time-to-live or timestamp.

The cache interface layer does not automatically write updates through to the repository. Instead, a separate function *DhtSync()* causes the cache to send an update request to synchronize the cached copy with that in the repository. This enables applications to tailor access to the cache for different styles of object access.

For example, by delaying synchronization and specifying the non-validation cache policy, the cache can be used as a local workspace, enabling a development style similar to that provided by NSE [1]. Objects, once placed into the cache, are read and updated locally, and thus are not affected by updates from other developers. A "sweep" application periodically synchronizes the cached copies, possibly invoking merging tools for objects that have changed in the interim.

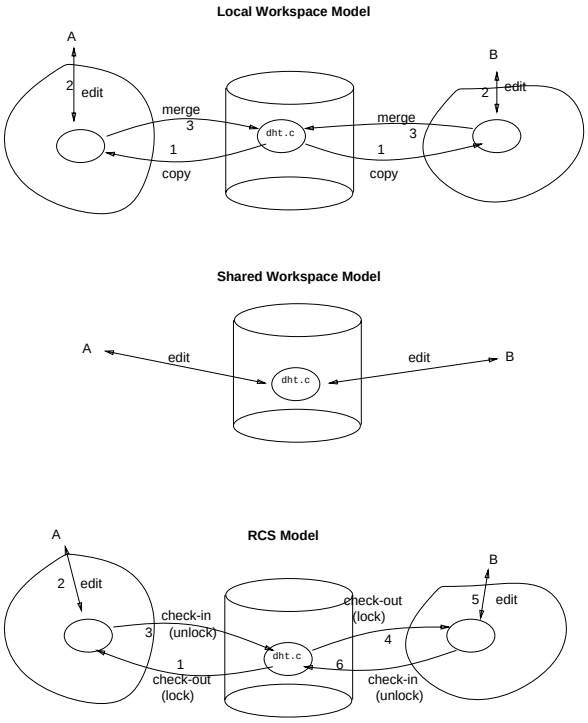

Figure 6: Software development workspace models.

Alternately, updates can be written-through immediately, by calling *DhtSync()* after each update operation. This, coupled with the verifying cache policy, can be used to implement a "shared workspace" style of development (Figure 6), in which each developer sees updates from other developers upon object access.

To simulate an RCS-style of development, in which developers obtain exclusive write access to an object through locking, a *lock* attribute must be added to objects, which is set to the user-id of the developer who seeks to lock the object. The DHT concurrency mechanism ensures that only one developer can set the lock, which is cleared when the object is "released". However, applications must cooperate by not modifying objects unless they have successfully set the lock attribute; there is no way to enforce the lock by denying updates if someone insists on updating an object. This style can be coupled with the validating or non-validating cache policy, depending on the preferences of the developer.

4 Incorporating Process Enactment

A software process is a partially ordered set of tasks that must be performed to develop software. A software process *model* is a description of a software process. If the description is sufficiently formalized, it is possible to execute process models for simulation, analysis, and *enactment*.

Software process enactment uses a formal description of a software process to guide, monitor, and control the process by having a process interpreter or engine execute a formal process description. The interpreter, embedded in a software development environment, performs three functions: guidance, monitoring, and control.

Guidance involves leading developers through the process by issuing prompts or notifications as to what tasks should be performed at a given time.

Monitoring allows managers and engineers to assess the current state and progress of the process.

Control means ensuring the process is followed by restricting developer actions to those that conform to the process description.

A process describes what steps need to be performed to develop products. At any given time, several products may be under development, so it is important for a process enactment mechanism to be able to keep track of multiple instances of a process simultaneously, and to be able to cope with the interactions among multiple processes executing concurrently.

For example, a software system may have several developers performing maintenance on different modules at the same time. This means that, for each module, an instance of a software maintenance process needs to be executed. Furthermore, different modules developed in different organizations that are part of a common system build may be constrained by different software process models as well.

A software process has a natural representation as a hypertext graph: the nodes represent tasks, node attributes can represent task pre- or post-condition values, while links specify the sequence in which the tasks should be performed; the resulting nodes and links can be browsed and followed just like other hypertext objects. Thus, processes can be enacted by browsing the hypertext.

The links between task nodes and product nodes (artifacts) change as tasks are performed and products are created or modified. In addition, the state of progress for a process instance must be maintained. Finally, enactment must take place using existing tools already familiar to users. Thus, we desire to have enactment in DHT work

with browsers already in place, rather than introduce some new process-specific browser.

A DHT hypertext process model contains three types of links:

1. *Process decomposition* links. These model the decomposition of high level tasks into lower level, smaller tasks, and eventually into primitive actions. These are static links that do not change unless the model itself is modified to correct errors or reflect changes in policy.

2. Task and action *precedence* links. These specify the order in which tasks should be performed. These are also static links that evolve slowly.

3. *Available task* links. These model the relationship between a particular product node and the tasks which should be performed on it at a given time, as specified by the process model. These are dynamic links that change as the model is enacted.

As an example, the following is an informal description of the process for modifying a module:

1. Retrieve module.

2. Edit module to implement changes.

3. Compile module.

4. Unit test module.

5. if the unit test is successful, create a new version of the module; otherwise,

6. Debug the module and return to step 1.

This description is a *model* of a process. At a given time, many *instances* of the process may be active, on different modules by different developers. Each instance has separate process *state* including the module begin modified, the developer doing the modification, the last step completed, etc. To support process enactment, it is necessary to keep track of this state for each process instance [9], in order to guide the developer through the process tasks in the appropriate sequence. This is the function of "available-task" links. Available-task links serve to notify developers that a task should be performed on an artifact by linking the product to a task node.

Process state is reflected in the state of the products (artifacts) that the process instance affects: when the "edit" task above is performed on a module, its state changes, as reflected by the changes to its contents and timestamp affected by the edit. A link source predicate can examine this state to establish a relationship between a product node and a task node. When the link's source predicate is true when applied to the product node, the link will resolve to a task node that should be performed on the product.

Task nodes can either be narrative descriptions of the task to be performed, or executable scripts that perform the details of the task. In the latter case, the link's resolution function passes the node to an interpreter to execute the script.

Figure 4 shows link specifications for the example module modification process.

The significance of this approach to modeling process instances is that the mechanism for process enactment is embedded entirely within the process representation, as the source predicates and resolution functions of available-task links. In contrast to other enactment systems which employ an environment [13], process-aware tools [23], or process state database [9] to execute process specifications, DHT process models can be enacted simply by browsing the process hypertext using any DHT browser or tool. This means that process enactment can be introduced into an existing environment with minimal disruption.

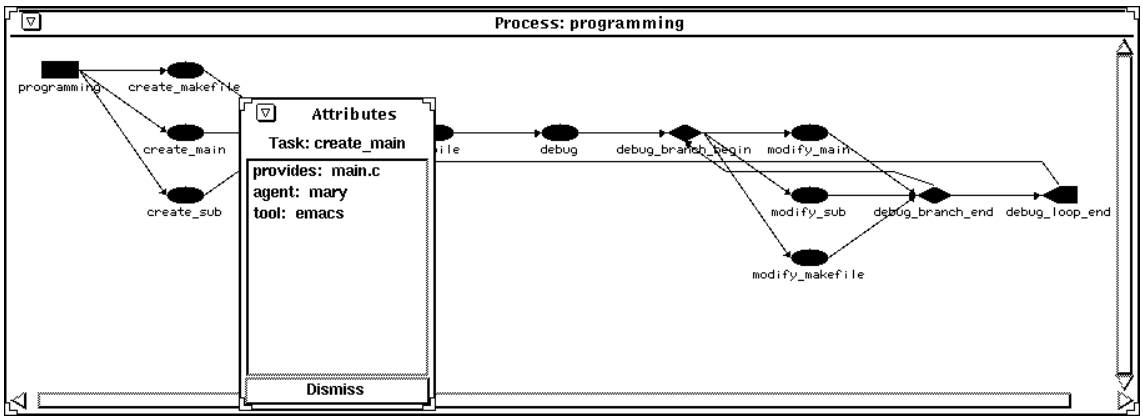

Figure 7: Example of a process model.

Attribute	Value
source	`{[type $node]==DHT:C}`
source_anchor	Global
dest	`{eval [get-contents process:edit-module]}`
dest_anchor	Global
type	DHT:Available-task

Available-task link for "edit" task.

Attribute	Value
source	`{[type $node]==DHT:C && [status $node]=="edited"}`
source_anchor	Global
dest	`{eval [get-contents process:compile]}`
dest_anchor	Global
type	DHT:Available-task

Available-task link for "compile" task.

Attribute	Value
source	`{[type $node]==DHT:C && [status $node]=="compiled"}`
source_anchor	Global
dest	process:unit-test-spec
dest_anchor	Global
type	DHT:Available-task

Available-task link for "unit-test" task.

Figure 8: Available task links.

5 Related Work

We have approached integration by providing the illusion of a central repository through the introduction of a layer between storage managers and users of data. Such a layer provides a logically integrated universe of data objects that conform to a hypertext data model. It also provides the physical integration of participating repositories necessary for users to access instances of data objects.

Other research uses the same general approach we have taken, but with different data models. For example, network file-system solutions [11, 19, 20], implement a common global file-system at the integration layer, where each repository exports its objects as files in a single unified directory tree. However, network file-systems are a lowest-common denominator solution [21]: the file/directory model lacks explicit constructs to represent the numerous relationships that exist among software artifacts [17]. As a result, ad-hoc techniques such as file naming conventions, and numerous tool-specific databases like Makefiles, tag files etc. are required to augment the basic directory-file model. Consequently, information sharing is at a rudimentary level.

The multidatabase approach [3] occupies the other extreme: here the integration layer provides a relational or object-oriented data model with explicit relationship types. This would seem to solve the relationship problem of the network file-system; both of these models have been used in conventional environments. However, the complexity of constructing and maintaining a single global schema that captures all of the concepts present in each participating repository, combined with the requirement that integrated repositories have database functionality, makes this approach costly and difficult to implement [6].

A number of research projects have applied hypertext to software object management, including the Hypertext Abstract Machine (HAM) [4], the Documents Integration Facility (DIF) [7], and HyperCASE [5]; however, these are based on a single, centralized repository architecture.

Notable exceptions are PROXHY [10] and Chimera [2]. In contrast to DHT, these systems focus on adding new links among existing artifacts; existing relationships are not translated into links.

Finally, ISHYS [8] and sigmaTrellis [22] explored process modeling and enactment via hypertext, although in a centralized architecture.

6 Conclusion

We began with the assertion that software development in the future will be performed by cooperating development teams. These teams will be autonomous, widely distributed, and loosely associated in "virtual" enterprises, yet will need to share data in the form of software artifacts, and the relationships among them, that form the products of the development process.

We observed that the conventional software environment architecture is inadequate to support virtual software enterprises, because it relies on a central repository to serve as the medium of data sharing among users; thus, we proposed to address a research problem: how do virtual enterprises share data among their dispersed, loosely-coupled and autonomous participants?

We proposed a solution based on providing an integration layer between autonomous software repositories and their users, that would provide the appearance of a central repository while maintaining the distributed physical environment. The integration layer achieves two levels of integration: *logical* integration by describing data and operations on them in terms of a common hypertext data model, and *physical* integration via a distributed architecture for access to

autonomous repositories.

The DHT approach has the following benefits:

Evolutionary approach to integration It is possible to migrate a conventional Unix toolbox oriented environment to DHT without recompiling any of the individual tools. This is because DHT offers several levels of tool integration, depending on the degree of "hypertext awareness" desired for a given tool.

This evolutionary approach to incorporating existing tools into a DHT environment enables tool integrators to make tradeoffs between integration effort and hypertext functionality. This gives administrators great flexibility to preserve existing investment in tools and training while simultaneously obtaining the advantage of DHT's integration and hypertext capabilities

Transparent Process Enactment By representing software processes as hypertext graphs, DHT achieves enactment without the need for an explicit process interpreter or environment. This is a significant advantage in a virtual enterprise, where each participant may already have a favorite development environment in-place. The DHT approach allows process enactment to co-exist with existing tools and environments.

Comprehensive solution The most significant contribution of DHT is the way it applies the features of hypertext to data integration, combining intrinsic support for user interaction with data modeling and access. DHT combines the advanced data modeling capabilities of semantic nets with the natural navigation-based access of file systems, and the intuitive direct-manipulation browsing features of hypertext. This means that as soon as a transformer is put in place to export data from a particular repository, the user interface and access operations to those data are also in place. Furthermore, both the user and access interface conform to a single common model, therefore maintaining highly transparent access to heterogeneous data. The result is effective yet low in implementation cost.

We set out to develop a solution to the problem of sharing software engineering data among distributed, autonomous development teams, with data modeling and management appropriate for software artifacts and relationships, transparent access to heterogeneous, autonomous legacy repositories, tool integration, process modeling and enactment, and low implementation cost. DHT achieves these goals by applying hypertext concepts to logical and physical integration. In so doing, DHT solves the practical problem of sharing data in a virtual enterprise, and establishes a basis for continuing research in integrating heterogeneous software object management repositories, data models, and implementation architectures using easily navigated wide-area hypertexts. The interested reader should consult [14] for further details and examples.

References

[1] Evan W. Adams, Masahiro Honda, and Terrence C. Miller. Object management in a CASE environment. In *Proceedings of the 11th International Conference on Software Engineering*. The Association for Computing Machinery, 1989.

[2] Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead, Jr,. Chimera: Hypertext for heterogeneous software environments. In *European Conference on Hypermedia Technology*, Edinburgh, Scotland, September 1994.

[3] M. W. Bright, A. R. Hurson, and Simin H. Pakzad. A taxonomy and

current issues in multidatabase systems. *IEEE Computer*, March 1992.

[4] Brad Campbell and Joseph M. Goodman. HAM: A general purpose hypertext abstract machine. *Communications of the ACM*, 31(7), July 1988.

[5] Jacob L. Cybulski and Karl Reed. A hypertext based software-engineering environment. *IEEE Software*, March 1992.

[6] D. Fang, J. Hammer, D. McLeod, and A. Si. Remote-exchange: An approach to controlled sharing among autonomous, heterogenous database systems. In *Proceedings of the IEEE Spring Compcon, San Francisco*. IEEE, February 1991.

[7] Pankaj K. Garg and Walt Scacchi. A hypertext system for software life cycle documents. *IEEE Software*, 7(3):90–99, May 1990.

[8] Pankaj K. Garg and Walt Scacchi. On designing intelligent software hypertext systems. *IEEE Expert*, 1990.

[9] Dennis Heimbigner. The ProcessWall: A process state server approach to process programming. In *Proceedings of the Fifth SIGSOFTSymposium on Software Development Environments, Tyson's Corner, Virginia*, December 1992.

[10] Charles J. Kacmar and John J. Leggett. PROXHY: A process-oriented extensible hypertext architecture. *ACM Transactions on Information Systems*, 9(4):399–420, October 1991.

[11] James Kistler and Mahadev Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–20, February 1992.

[12] Henry F. Korth and Abraham Silbershatz. *Database System Concepts*. McGraw-Hill, 1986.

[13] Pei-Wei Mi and Walt Scacchi. Process integration in CASE environments. *IEEE Software*, 9(2):45–54, March 1992.

[14] John Noll. *Software Object Management in Heterogeneous, Autonomous Environments: A Hypertext Approach*. PhD thesis, University of Southern California, September 1995.

[15] John Noll and Walt Scacchi. Integrating diverse information repositories: A distributed hypertext approach. *IEEE Computer*, 24(12):38–45, December 1991.

[16] John Noll and Walt Scacchi. A hypertext system for integrating heterogeneous, autonomous software repositories. In *Proceedings of the Fourth Irvine Software Symposium*, pages 49–59, Irvine, CA, April 1994. Irvine Research Unit in Software and ACM/SIGSOFT.

[17] Maria H. Penedo, Erhard Ploedereder, and Ian Thomas. Object management issues for software engineering environments; workshop report. In *SIGSOFT '88: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston*. The Association for Computing Machinery, 1988.

[18] P. J. Plauger. *The Standard C Library*. Prentice Hall, 1992.

[19] Herman C. Rao and Larry L. Peterson. Accessing files in an internet: the jade file system. *IEEE Transactions on Software Engineering*, 19(6):613–625, June 1993.

[20] Mahadev Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, 18(1):1–9, January 1992.

[21] Peter Scheurmann, Clement Yu, Ahmed Elmagarmid, Ĥector Garcia-Molina,

Frank Manola, Dennis McLeod, Ãrnon Rosenthal, and Marjorie Templeton. Report on the workshop on heterogeneous database systems. *SIGMOD Record*, 19(4), December 1990.

[22] P. David Stotts. sigmaTrellis: Process models as multi-reader collaborative hyperdocuments. In *Proceedings of the Ninth International Software Process Workshop, Airlie, Virginia*, October 1994.

[23] Richard Taylor, Frank Belz, Lori Clarke, Leon Osterweil, Richard Selby, Jack Wileden, Alexander Wolf, and Mical Young. Foundations for the arcadia environment architecture. In *SIGSOFT '88: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston*. SIGSOFT/SIGPLAN, November 1988. Available in SIGSOFT Software Engineering Notes 13(5) and SIGPLAN Notices 24(2).