An Overview of Repository Technology

Philip A. Bernstein[1]
Digital Equipment Corp.

Umeshwar Dayal[2]
Hewlett-Packard Labs

Abstract

A repository is a shared database of information about engineered artifacts. We define a repository manager to be a database application that supports checkout/checkin, version and configuration management, notification, context management, and workflow control. Since the main value of a repository is in the tools that use it, we discuss technical issues of integrating tools with repositories. We also discuss how to implement a repository manager by layering it on a DBMS, focusing especially on issues of programming interface, performance, distribution, and interoperability.

1 Introduction

Metadata management is a growing part of the database business, driven by many trends in information technology. For example,

- Business process re-engineering has gotten the attention of executives of most large enterprises. This leads to the development of large process models and information models, which are metadata and need to be managed.
- Information technology departments are buying computer-aided software engineering (CASE) tools piecemeal, and are now finding it difficult to merge the design data (metadata) developed with each of them.
- Large enterprises are deploying "data warehouses" to cache databases in user-oriented form. Tracking this data and where it comes from is a metadata problem.
- Diverse database types, managed by heterogeneous database systems (DBMSs), some not managed in a DBMS at all, are being widely deployed. Their descriptions need to be managed in an integrated way.
- Users want object-oriented (OO) application development. Many of the objects produced by OO development are metadata, such as interface descriptions and class hierarchies.
- Manufacturing enterprises are alarmed by the high cost of information technology to introduce new products, most of it just translating information between different formats. They need to manage these data formats and the data they describe, so that all their information management, computer-aided design (CAD), and product data management tools can share on-line databases.

The technical requirement to support the above activities is not just a fancier catalog for the customer's database. Nor is it just introducing an object-oriented DBMS in place of a relational one. Rather, it's implementing a layer of control services on top of the DBMS, called a repository manager, and integrating it with many tools. The result of this integration is a framework for metadata management, called a *repository system.*

There is a small but rapidly-growing market for repository systems. This market is led by vendors of data dictionaries, CASE tools and CAD tools. Database researchers have had little influence, except indirectly via their OODB work. Conversely, the vendors have made little use of what has been learned about OODBMSs. This paper attempts to bridge this gap.

We propose a definition of repository (Section 2). We then discuss what functions a repository manager must support (Section 3), how to integrate tools with a repository (Section 4), how to layer a repository system on a DBMS (Section 5), and what technical problems need to be addressed for this technology to move forward (Section 6).

Proceedings of the 20th VLDB Conference
Santiago Chile

[1] Current Address: Microsoft Corp., One Microsoft Way, Redmond, WA 98052-6399. philbe@microsoft.com

[2] Address: Hewlett-Packard Labs, 1501 Page Mill Road (1U-4), P.O. Box 10490, Palo Alto, CA 94303-0969. dayal@hplabs.hp.com.

2 What's a Repository?

A *repository* is a shared database of information about engineered artifacts produced or used by an enterprise. Examples of such artifacts include software, documents, maps, information systems, and discrete manufactured components and systems (e.g., electronic circuits, airplanes, automobiles, industrial plants). The fact that artifacts are "engineered" leads to a variety of requirements for database structuring and access, which we describe here and in Section 3.

Over the lifecycle of engineered artifacts, many objects of many different types are defined, created, manipulated, and managed by a variety of tools that need to share data. For example, software engineers use design tools, language editors, compilers, builders, and debuggers to create and test programs. Project managers use planning, tracking, financial analysis, and reporting tools to create and manage project plans, spreadsheets, and reports. Technical writers use text and graphics editors, hypertext and document management tools to produce, manage, and manipulate compound documents. System managers use monitoring, reporting, diagnosis, and configuration tools to monitor and reconfigure system components.

The objects themselves may be stored in a variety of storage systems, such as file systems, database systems, or hard-copy filing cabinets. Descriptions of these objects are stored in the repository. In addition, the repository may store information about an object's location, its revision history, the tools and processes that were used to build it, constraints that it satisfies, who is authorized to access or modify it, who is responsible for managing it, and its dependencies on other objects.

Storing this information in a common repository has several benefits. First, since the repository provides storage services, tool developers need not create tool-specific databases of the objects that tools or use.

Second, a common repository allows tools to share information so they can work together. For example, the repository can store meta-metadata (such as allowable data formats for record and field definitions), metadata (such as specific record and field definitions), and data itself, which may be shared by tools such as compilers and debuggers, database query processors, forms managers, and report writers. Without a common repository, special protocols would be needed for exchanging information between tools. By conforming to a common *data model* (i.e., allowable data formats) and *information model* (i.e., schema expressed in the data model), tools can share data and metadata without being knowledgeable about the internals of other tools. In this respect, a repository is like a data dictionary. However, while data dictionaries typically only store metadata (database schemas, record and field definitions), a repository can store information about the whole range of object types pertinent to an enterprise.

Third, the information in the repository is subject to common control services, which makes sets of tools easier to use. Since a repository *is* a database, it is subject to database controls, such as integrity, concurrency, and access control. However, in addition, a repository system provides checkout/checkin, version and configuration control, notification, context management, and workflow. These services are described in the next section. Of course, even in the absence of tools sharing data, a single tool can benefit from these repository services, though one would probably not invest in building an independent set of complete and robust repository services unless the investment were amortized over many tools that would use the services (even if they won't share via the repository).

By promoting data sharing through common data and information models, and imposing a common set of control services, a repository is the centerpiece of an integrated environment in which a dynamic collection of tools can work together.

3 Repository Manager Functions

3.1 Introduction

A *repository manager* provides services for modeling, retrieving, and managing the objects in a repository. It typically uses one or more storage managers to store the objects. For example, a file object (e.g., a document, circuit diagram, or source code program) may be stored in a file system, while descriptive attributes of the file object (e.g., how and when it was created, who owns it, where to find it, lists of related objects) may be stored in a DBMS. When a user asks to retrieve an object, the repository manager looks up the object's location attribute and then copies the object into the user's workspace, such as a file directory or a private database.

A repository manager should provide the standard amenities of a DBMS: a data model (to structure a repository), queries (to browse a repository), views (to enhance data independence of tools that access a repository), integrity control (to trap integrity violations), access control (for secure access), and transactions (for atomic multi-statement updates). Managing metadata requires additional services beyond those of a conventional DBMS. These services are the main added value of a repository manager: checkout/checkin, version control, configuration control, notification, context management, and workflow control. We *define* a repository manager to be a data manager that offers these functions. Most repository manager products offer these functions, though not always in their full generality. We describe these functions below.

3.2 Checkout/checkin

Activities that use a repository can be of long duration, lasting days or months. Treating the entire activity as one transaction is impractical. For example, a system crash during such an activity could lose days of work.

Therefore, the repository manager must support *checkout* and *checkin* of objects. The checkout operation copies the object from the shared repository into the user's private workspace. After working on the object, the user issues a checkin operation, which copies the object from the private workspace into the shared repository. Checkout and checkin execute as (separate) short transactions. Essentially, checkout sets a persistent lock on the object, which is released by checkin. Checkout should support shared and exclusive modes.

3.3 Version control

Repository objects typically undergo a series of revisions, which the repository manager represents as versions. A *version* is a semantically meaningful snapshot of an object at some point in its lifecycle. The repository manager maintains each object's version history. A version history is a directed graph with one node per version and an edge from version A to version B if B was derived from A. Features of a version model should include [7]:

- A mechanism for representing versions as objects in the repository.
- A version naming mechanism, preferably both with default and user-supplied names.
- An operation for deriving a new version from an old one. The derivation operation specifies whether the properties and behavior of the new version are copied from the old version or are modified in some way.
- Constraints on the version history. Some models restrict the version history to a single path, but this is too restrictive for most applications.
- A mechanism for identifying a particular version to be used in a given context.
- The semantics of checkout and checkin of versioned objects. Checkout in exclusive mode might automatically create a new version, while checkout in shared mode does not. On checkin, the model might allow multiple branches, or might require that concurrently created siblings be merged.
- A semantics for relationships between versioned objects. A new version of an object may inherit relationships that were attached to the previous version of the object.
- An operation for declaring that two or more independently-developed versions merge into one version. The operation must specify how the properties and behavior of the new version are derived from those of the versions being merged (inherit from only one of them, or perform some semantically meaningful merge). Some version models distinguish one path in the version history as the "main" line of descent; other versions are variants which are merged back into the main line from time to time. While this main-line concept is sometimes useful, a general version model is needed that supports true alternatives that may never be merged.

3.4 Configuration control

Some objects in a repository are hierarchical collections of other objects, called *composite objects*. Both a composite object and its components may be versioned. A *configuration* is a binding between a version of a composite object and a version of each of its (versioned) components. Not all the components of a configuration need to be versioned, e.g., the authors in a versioned author list. Features of a configuration control model should include:

- A mechanism for representing a configuration.
- A mechanism for identifying a configuration's component versions, either explicitly by name or implicitly by context (e.g., John's current configuration of A includes the last checked-in version of B and the last version of C created by John).
- Operations for attaching component versions to configurations and for detaching them.
- Mechanisms to define and enforce constraints on configurations. One should avoid fixed constraints on all configurations. E.g., a model might require that a configuration contain only one version of each component. This is useful for software systems, where different versions of a component could interfere with each other. But it's inconvenient for complex vehicles, which might contain more than one version of a part, especially after being repaired a few times.
- The semantics of change propagation in configurations. If one version of a component is replaced by another version, does that automatically result in a new configuration? If so, then when a versioned object is checked out, its composite objects must also be checked out, reducing concurrency. It can recursively proliferate configurations up the composite object hierarchy; e.g., replacing a version of a screw would cause a new version of a vehicle to be created. Not all applications want this behavior, so the model should allow the configuration's definer to specify the desired semantics for each component: create a new version, do not create a new version, or perform

some other action (e.g., notify a user or invoke a tool).

3.5 Notification

Many objects in a repository are interrelated. When an operation is applied to one object, operations may need to be applied to related objects. For example, when a source module is changed, rebuild dependent object modules. When one representation of a design object is changed, update the other representations. When a checkout request is received for an already-checked-out object, grant the request but notify the concurrent users of the object.

These are all examples of *notification*, where the repository manager implements rules of the form "When event E occurs, if condition C holds, then perform action A." The rules for change propagation, version control, and configuration control might be "hardwired" into a repository manager. However, a general facility for users to define and implement rules would be more flexible. It would allow customization of version and configuration control. It would support integrity control, access control, and propagation of operations to related objects. It would also enable the definition of enterprise-specific, domain-specific, or application-specific policies. For example, release control might define the rule "When the last signature on the approval list is obtained, change the status of the object to 'Released'."

3.6 Context management

A context defines a view of the objects in the repository. It is typically used to define the set of objects that an engineer is manipulating for a particular task. It may also include user preferences (e.g., language, editor, display), and specific rules and constraints to be enforced. One should be able to put arbitrary objects in a context, not just files. Thus, a file directory is often an inadequate mechanism for context management.

To carry out a task, a user *opens* a context and then performs operations on objects visible in this context. When the task is finished, the user *closes* the context. A user can have many contexts open at a time (e.g., an edit context and a compile context). Also, contexts can remain open for a long time, and therefore should be persistent. This allows a user to leave a task for awhile, and the same or different user pick it up later.

3.7 Workflow control

Engineered artifacts progress through phases of a lifecycle, such as requirements, specification, design, analysis, production, testing, and release. A repository manager should support a *workflow control model* to track an object's state relative to its lifecycle. A *promote* or *demote* operation changes the state of the object. It can either be invoked manually by a user or automatically by a notification rule (e.g., because a test ran successfully).

A more general workflow control model would provide primitives for describing a long duration *activity* as a collection of *steps*, the flow of control and data among the steps, steps for handling exceptions, etc., and a controller to drive executions of activities. Such a model is of general value, not just to the design of engineered artifacts, so it should be available separate from a repository manager. However, it may use a repository to hold its metadata.

4 Tool Integration

Some users just want a repository for its data modeling capability, for example, to describe scientific data sets. However, most users don't really want just a repository *per se*. Rather, they want tools, and the repository makes those tools more valuable in some way.

Even when only one tool is of interest, users often want (some of) the control services that a repository manager provides. They'll accept the database (i.e., repository) that comes along with it, if it's explained why they need it: to model objects and collections for configuration management, to model relationships for notification services, etc.

Most users quickly graduate to an environment where multiple tools are used. For example, they might want toolsets for business process modeling, application design and analysis, DB design and analysis, application programming, application re-engineering, product release management, or computer systems management. At this point, they want their toolset integrated. There are a number of dimensions in which to integrate a toolset. One critical dimension is to have the tools share data. Data sharing among tools requires a repository.

A minimal level of repository integration is *tool invocation*. A tool is defined in the repository by its interface and invocation method, so the repository manager can invoke it and pass it objects of the correct types. When invoking the tool, the repository manager can exercise some control, such as triggering notifications or updating workflow state. At this level of integration, though tools are known to the repository, they may not be able to share data.

One way for tools to share data is via *data exchange*. Data exported by one tool is translated into the import format of another tool. With *n* tools, you need n^2 translators. One can do better by defining a canonical format for data translation, and building two translators for each tool, to translate the tool's export format to canonical form and to translate canonical form into the tool's import format. This reduces the problem to *2n* translators. The main technical problems are to have a sufficiently rich data model to represent all shared data, a complete and tool-neutral information model for objects that tools want to share, and translators that properly interpret the semantics of the data they translate. Some examples of this *data exchange switch* approach are the Express data interchange standard for CAD data translation [12], and the Software One Exchange product that moves data between CASE tools. This is a state-of-the-practice solution in many fields [16].

Although a data exchange switch can help tools share data, it is not a database system. By using a repository (i.e., database) to store data being shared by tools, one gets some additional benefits:

- you know where to look for objects. They're in the repository, not scattered around in files that are independently maintained by different tools.
- you only have one copy of each shared object, thereby avoiding inconsistencies between copies of the same object managed by different tools
- you don't lose information moving from one tool to another. Even data that can't be represented in the canonical model can at least be stored in the repository in tool-specific format and not simply lost during data exchange.
- you control all shared objects the same way, with the same version model, configuration model, etc.
- you can incrementally update shared data. By comparison, data exchange is, by its nature, a batch process.
- you can query the data, e.g., to find the revision history of an object or all dependent objects of an object.

Integrating a tool with a repository can be the same as integrating it with a data exchange switch: write translators that move data from tool export format to repository format and from repository format to tool import format. As with a data exchange switch, this requires that the repository have a rich canonical model that can represent all shared data and that the translators properly interpret the semantics of shared data. This approach works well when a tool checks out all the data it needs before executing, and checks in all the data it used when it finishes.

Some tools need to access data interactively during their execution (for example, a system builder, such as "make"). The batch translator approach, which imports and exports tools' data, doesn't work well in this case. One could modify each tool to directly access individual repository objects, as needed. However, this requires extensive modification of each tool, to access data in the repository's format using the repository manager's interface, which is quite expensive. Also, it results in a tool that is completely dependent on the repository manager, which limits the tool vendor's market to customers that use this repository manager. Instead, one can leave the tool unchanged and use a virtual repository interface, which traps all of the tool's data accesses (using whatever data access interface the tool depends on) and translates them into accesses of repository objects. This is a common way to integrate unmodified UNIX tools with a repository; all UNIX files operations are trapped and directed to the repository [8].

Since tools are what give value to a repository, it should be cheap and easy to integrate a tool with a repository, so many tools can be integrated. The state-of-the-art here is not very good and would benefit from some serious research attention. Today, integrating a tool with a repository is an art, requiring protracted technical negotiation between the tool vendors who want to share data and the repository vendor.

Tool integration work should be reusable, so a tool vendor's integration effort is portable to many repository managers. This requires that all repositories support the same application programming interface (API) and canonical information model. In practice, this is hard to do, since international standards are incomplete or immature and there is no dominant vendor dictating a standard. Still, there have been some successes. For example, in the discrete manufacturing area, there is a canonical information model (STEP [11]) which is written in the standard data modeling language (EXPRESS [13]), but the API for accessing such models (SDAI [14]) is immature and not yet widely supported. In the CASE area, there is growing support for the CASE Data Interchange Format (CDIF), which is an information model for CASE data, but this model is unrelated to the Portable Common Tools Environment (PCTE) API standard for CASE frameworks [10]. Moreover, PCTE is principally oriented toward version and configuration control of coarse-grained objects, and is unsuitable for fine-grained data sharing. Competing CASE APIs are also being considered. For example, ATIS (A Tool Integration Standard) is being discussed in ANSI [5]. ATIS is an extensible object-oriented information model covering the basic repository functions described in Section 3 plus an object-at-a-time API to access repository objects. The existing ISO standard for information resources is based on SQL, but seems to have had little commercial impact[1]. Presented with such a confusing picture, many tool vendors are delaying their investment in repository integration until the market stabilizes. Or, they have implemented their

[1] Incidentally, all of the API standards mentioned above (SDAI, PCTE, ATIS) are throwbacks to a CODASYL-like model. They offer record-at-a-time access will minimal content-based retrieval and no attempt at SQL compatibility. They would benefit from attention by DBMS language experts.

own information model which at least ensures their own tools can interoperate (e.g., Texas Instruments' Information Engineering Facility (IEF)).

There's a temptation to have the repository be the tool's native storage manager. While this is feasible in principle, it is often undesirable, because high performance may require that the tool use its private repository format. Most tools manipulate objects in a "main memory database" which they construct after checking out the required files. A repository manager is inherently slower, in many cases too slow for interactive use.

Even if a tool *T* uses a repository manager as its storage manager, integrating *T* is still an issue with tools that don't use this repository manager. To integrate *T* with tool *U*, either *U* must integrate with *T*'s repository manager or *T*'s and *U*'s repository managers must interoperate (see Section 5.5).

5 Repository Manager Implementation

A repository manager is an application layered on a DBMS. In this section we discuss some of the issues involved in implementing this application.

5.1 Repository API

The API requirements for a repository manager are essentially the same as those for DBMSs in support of CAD. These requirements are met well by OO APIs [1], which allow you to:

- construct types for objects that support repository control functions, such as versions, version histories, configurations, contexts, and rules.
- construct new atomic and complex types that represent objects within their domain. You can also construct bulk types of these objects, such as tuples, sets, sequences, and lists.
- represent relationships in a flexible and natural way.
- use a inheritance hierarchy to share type information across multiple types.
- incorporate type-specific operations that encapsulate tools (e.g., *edit* a document, *approve* a design).
- navigate among objects, an object-at-a-time.
- dynamically construct new objects.

Queries over collections of objects in the repository are needed in addition to object-at-a-time navigation. Highly functional query languages are starting to be developed for OODBMSs [2], but they are not yet ubiquitous in OODBMS products.

Entity-relationship APIs support the data modeling requirements of repositories, but they don't allow you to add new operations. SQL has traditionally not supported many of the above capabilities. However, with the addition of abstract data type facilities in many relational DBMSs, there may soon be little difference in functional capability between OO APIs and SQL beyond syntactic sugar [3,9].

5.2 Database Engine Support

A repository manager is an application layered on a database engine, which could be a file system, relational DBMS (RDBMS), or OODBMS.

Most repository managers use a file system to store coarse-grained objects, such as program source, text, and diagrams. Some repository managers also use files to store objects that support their control functions, such as versions and relationships. For example, classical CASE repositories operate this way, such as CMS, SCCS, rcs, MMS, and make. Although file systems are light on functionality compared to DBMSs, they do offer two advantages: they're ubiquitous, so by relying on a file system, a repository manager can easily be ported to many operating systems; and they offer excellent performance for sequential access.

Some repository managers are implemented on a combination of RDBMS and file system. The RDBMS tables store descriptions of objects, and files store objects themselves. This allows one to use unmodified tools that use files for object storage, while for description data one gets the benefits of DBMS amenities, such as transactions, referential integrity, and queries. However, there are some problems with this approach: Updates to descriptions of objects and objects themselves cannot be grouped in the same transaction (because file systems don't support transactions). Administration of objects and descriptions is hard to coordinate (e.g., to coordinate backup and recovery so that objects and descriptions can always be recovered to a mutually consistent state). And one cannot build indices on objects and execute queries on objects (unless one replicates some of each object in its RDBMS description, which creates potential consistency problems).

The first two problems can be solved by storing objects in the RDBMS as binary large objects (BLOBs). The last requires that objects be decomposed into pieces which are stored separately in the RDBMS. But this is hard to do with conventional RDBMSs, which require objects to be laid out in rigid table structures.

Ideally, a repository manager would map its "object base" into labeled directed graphs, where objects are mapped to nodes and relationships are mapped to edges. Operations include object-at-a-time navigation, following paths of objects, and taking transitive closures of subgraphs. A growing number of RDBMSs are supporting graph-type databases via OO features, such as user-defined data types, type constructors for complex user-defined types (such as records and arrays), and extensions to SQL for transitive-closure-type operations. The SQL standard is also evolving to support these features [9].

OODBMSs are a promising target for repository manager implementation, because they can directly implement graphs and graph operations. They have rich facilities for user-defined types and type constructors. They can lay out complex objects in contiguous memory instead of splitting them into different tables. They can execute type-specific methods. They are optimized for navigational operations; in some products, edge traversal in the database graph only costs a pointer dereference, either by directly mapping database pages into main memory or by "swizzling" (mapping) disk pointers into memory pointers when objects move to memory. Also, since most OODBMSs target the CASE and CAD markets, they provide limited forms of the repository control functions (typically checkout/checkin, versions, and configurations).

However, most OODBMSs are less mature than RDBMSs. Many provide limited transaction facilities (e.g. medium- or coarse-grained locking), limited support for queries, views, constraints, and triggers, and weak subsets of SQL with limited query optimization.

In summary, both RDBMSs with OO features and the best OODBMSs are promising targets for repository manager implementation. Since both types of DBMS products are immature and since experience in building repository managers on such types of products is limited, it is too soon to say which type will dominate in the long term.

5.3 Performance

A critical problem of today's repository managers is poor performance. A checkout or checkin operation on a complex design object can take tens of minutes, for example, to traverse a large object base to find the relevant versions of objects in a large configuration. A design tool that accesses large parts of a repository can take hours. Users find this barely acceptable. They often work around the problem by storing a large set of objects as one large object whose internal structure isn't visible to the repository manager and which is read and written as a unit. Of course, this reduces the value of many repository control services, since they are unable to help manage the fine-grained components of a design.

One aspect of repository management performance is cache management. Most RDBMSs implement a record (i.e., object) server, and the client process (in this case the repository manager) has an object cache that is invalidated on every transaction commit. Thus, each object access costs a client-server message, except when the server sends multiple useful objects in bulk (e.g., for set-oriented access) or when the client's transaction accesses the same object multiple times. Most RDBMSs support "stored procedures," which are application procedures that run in the object server. One client call to a stored procedure can result in many object accesses, thereby reducing client-server traffic.

Most OODBMSs implement a page server and the client process (the repository manager) has a page cache. Since a page access brings many objects to the client, if the client accesses many objects on a page (e.g., navigating an object-at-a-time), client-server traffic is lower than with an object server. Moreover, in some OODBMSs, the client cache is not invalidated on every transaction commit. Thus, when the client runs many transactions, it can build up a useful cache that's continually reused, further reducing client-server traffic. This gives OODBMSs a performance advantage, at the cost of protection between the repository manager's and OODBMS's address space.

Many customers insist that the repository manager be layered on the same DBMS they use for other purposes, e.g., for easier administration and training. This means the repository manager must be portable across DBMSs, which makes it difficult to get the full performance benefit from certain DBMSs. We predict that successful repository manager vendors will attain this portability with high performance, but it entails much engineering expense and therefore isn't around the corner.

OODB benchmarks are probably representative of how repository managers use a DBMS [6], but we know of no published workload analyses to substantiate this intuition. Even less is known of how a repository manager's *use* of a DBMS affects performance. This area would benefit greatly from systematic study.

5.4 Distribution Issues

A repository manager may offer transparent access to distributed data. That is, a client application may issue a location-transparent access, which the repository manager translates to a local access on the appropriate repository manager server. If the repository manager is implemented on a DBMS that supports transparent distribution, then it can trivially rely on the DBMS's capability. Otherwise, it must implement the capability itself.

In a distributed repository, objects in one repository may reference objects in other repositories. Since each repository manager needs the flexibility to move and delete its own objects, these references should be logical, not physical, and certain update operations need to check the integrity of these references. For example, if it is illegal to delete an object that participates in a relationship, then a delete needs to check the validity of relationships with objects in other repository managers.

Another form of integrity involves distributed transactions. If an update transaction accesses two or more repository managers, then those repository managers must use two-phase commit to ensure that the transaction is all-or-nothing. If the underlying DBMSs do not support this capability, then the repository manager needs a private workaround.

In a design environment such as CASE or CAD, users normally operate on repository data for long periods. Even if they use data managed by a remote repository manager, they probably need a private (replicated) copy of that data to get adequate performance. Fully symmetric data replication is beyond the state-of-the-art of distributed DBMSs. Therefore, one needs to exploit application-specific behavior to implement a replication scheme. For example, if versions are immutable, and sharing over short periods is rare, then copies of versions of popular objects can be distributed to all servers, say overnight, so they can be read locally on demand [15]. As another example, a remote checkout operation with intent to update may create a new locked version in the remote *and local* repository. On checkin, the local copy is written to the remote repository and deleted from the local repository. This replication technique is used in Digital's CDD/Repository [4].

5.5 Interoperation

To share information, heterogeneous repository managers have to interoperate. That is, object types in one repository's information model or data model must be accessible as object types in the other's model. This requires mapping operations on one repository manager into operations on the other (see fig. 1).

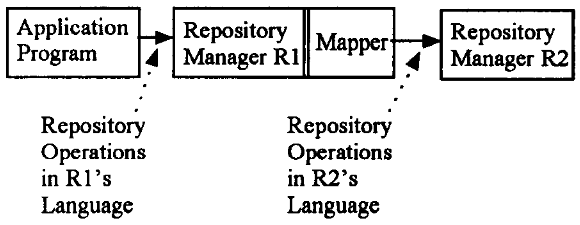
Application Program
Repository Manager R1
Mapper
Repository Manager R2
Repository Operations in R1's Language
Repository Operations in R2's Language

Figure 1: Mapping Operations Between Repositories

Ideally, each object exists in only one repository, with references to it in other repositories that share the object. This way, each object is only updated in one place. If replicas of an object exist in heterogeneous repositories, then updates must be propagated to all replicas. Semantic differences can make this hard to automate. Often, even a manual solution is hard, i.e., writing a repository-specific and information-model-specific program for cross-posting updates between two repositories. For example, if a versioned record definition in a CASE repository may be shared with an RDBMS catalog, how do you translate an update in the CASE repository that creates a new version into an RDBMS catalog update? What if they don't support the same data types? What if a relationship is many-to-one in one repository but many-to-many in another? Etc.

We believe it is unavoidable that many tools will, for the foreseeable future, have replicated heterogeneous repositories, for the following reasons:

- Many existing tools have already committed to a private repository implementation. E.g., database systems. These repositories are already well-tuned to the tool's performance requirements.
- Many tools need to be portable across operating systems. Therefore, they can only depend on a repository manager that runs on those operating systems and there are few such products on the market.
- In an object-oriented world, some objects will be designed to maintain some state that describes the object. It will be some time before repository technology is so mature that all objects will entrust all their state to a shared repository manager.

Thus, the problem of maintaining consistent heterogeneous repositories must be faced. Or we will have to wait for a repository technology to dominate the product world and for tools to be written or re-written to use that technology

6 Conclusion

We have argued that repository systems are an important type of database application and a worthwhile area of study. We proposed definitions for "repository" and "repository manager." We discussed approaches to repository tool integration. And we discussed repository manager implementation issues.

We believe repositories are a field that would benefit by more intense study by the database research community. Some specific areas that warrant attention are tool integration techniques, repository manager performance, a completely general model of versions and configurations, interoperability of heterogeneous repositories and repository managers, comparisons of commercial products, and case studies of using repositories in different tool domains.

7 Acknowledgments

Most of what we know about repository systems we learned from dozens of engineers at Digital who work on repository products and strategy. We especially thank Jonathan Bauer, Jim Gray, Ken Moore (now at Iris Software), Chip Nylander, Neil Schutzman, Al Simons, and Melissa Waldie.

8 References

1. Atkinson, M. et al., "The Object-Oriented Database Systems Manifesto," in *Deductive and Object-Oriented Databases,* Elsevere Science Publishers, Amsterdam, Netherlands, 1990.

2. Cattell, R.G.G. (ed.), *The Object Database Standard ODMG-93,* Morgan Kaufmann Pubs., 1993.

3. Committee for Advanced Database Function, "Third Generation Data Base System Manifesto," *ACM SIGMOD Record 19, 3* (Sept 1990), pp. 31-44.

4. Digital Equipment Corporation, *CDD/Repository Architecture Manual,* Field Test 3 Draft, March 1991.

5. Goering, R., "Standardization Effort Targets Data Management for CASE," *Computer Design 27, 18* (Oct. 1, 1988), pp. 28-30.

6. Gray, J. *The Benchmark Handbook for Database and Transaction Processing Systems,* Morgan Kaufmann, San Mateo, CA, 1991.

7. Katz, R.H. *"Toward a Unified Framework for Version Modeling in Engineering Databases."* ACM Surveys Vol. 22, No. 4}, December 1990.

8. Levin, Roy and Paul R. McJones, *The Vesta Approach to Precise Configuration of Large Software Systems,* Tech. Report 105, Digital System Research Lab, Palo Alto, June 1993.

9. Melton, Jim (ed.), *Database Language SQL 3,* ISO/ANSI Working Draft, ANSI X3H2-93-091 and ISO DBL-YOK 003, February, 1993.

10. Portable Common Tool Environment (PCTE) Abstract Specification. ECMA European Computer Manufacturing Association Standard ECMA-149. December 1990.

11. *Product Data Representation and Exchange. STEP Part 1: Overview and Fundamental Principles,* ISO CD 10303-1(E), International Organization for Standardization, 1993.

12. *Product Data Representation and Exchange. STEP Part 21: Clear Text Encoding of the Exchange Structure (Physical File),* ISO CD 10303-21, International Organization for Standardization, 1993.

13. *Product Data Representation and Exchange. STEP Part 11: Express Language Reference Manual,* ISO CD 10303-11, International Organization for Standardization, 1993.

14. *Product Data Representation and Exchange. STEP Part 22: Standard Data Access Interface Specification,* ISO WD 10303-22, Working Draft TC184/SC4/WG7/N350, International Organization for Standardization, 1993.

15. Prusker, Francis, Edward P. Wobber, "The Siphon: Managing Distant Replication Repositories," Technical Report 42, Digital Systems Research Center, Palo Alto, April 1990.

16. Ring, K. "U.K. Start-Up Software One Claims to Have Nuts and Bolts of AD/Cycle," *Computergram International,* Issue No. 1512 (Sept. 14, 1990), Applied Data Services, England.