

Improving The Interoperability of The Repository In a Box (RIB) Toolkit

Yuanlei Zhang

April 21, 2004

Computer Science Department

University of Tennessee

Knoxville, TN 37996-1301

ABSTRACT

Repository In a Box (RIB) is a software package for creating WWW metadata repositories. Interoperation, one of the most significant features of RIB, allows repositories using similar data models to share metadata with each other. Interoperation had very limited capabilities in previous implementations. This report presents ideas on how to improve the interoperability of RIB as well as their implementation. As a result of these improvements, application areas of interoperation will be broadened, and the improved interoperability will provide a general framework for effectively sharing metadata at various granularities, ranging from an entire repository to individual objects. Improving the interoperability of RIB will be a continuous work, with goals that include creating user-customized features, meeting specific user needs, and making the interoperation among repositories more powerful and more useful.

TABLE OF CONTENTS

Terminology

1. Introduction

2. Methodology

2.1 Improving the “interoperation feature”

2.1.1 Storing interoperation objects

2.1.2 Updating interoperation objects

2.1.3 Interoperation objects in the administration interface

2.1.4 Interoperation objects in the catalog

2.1.5 Deleting interoperation objects

2.2 Improving the “join feature”

2.2.1 Design concerns

2.2.2 Retrieving the content of the remote object

2.2.3 Replacing retrieved values

2.2.4 Clearing temporary values

2.3 Improving the “import feature”

2.3.1 Design concerns

2.3.2 Importing objects

2.3.3 Updating relationship fields

3. Conclusions

4. Applications

5. Future work

Acknowledgement

References

TERMINOLOGY

<i>Attribute</i>	A characteristic of a class for which a value may be given for an instance of that class
<i>Class</i>	Type of objects of the same kind
<i>Imported object</i>	Object created in the local repository as a result of the importing that is a static copy of the remote object
<i>Interoperation object</i>	Object created in the local repository as a result of the interoperation that is a “pointer” to the remote object
<i>Local object</i>	Object found in the local repository
<i>Local repository</i>	Repository currently being administrated
<i>Metadata</i>	Information that describes reusable objects, such as software
<i>Object</i>	An instance of a class
<i>Primary class</i>	Class whose objects are displayed in the catalog table of contents
<i>Relationship</i>	A relationship between two classes, expressed by a link from an instance of the first class to an instance of the second
<i>Remote object</i>	Object found in the remote repository
<i>Remote repository</i>	Repository that the local repository is interoperating with or importing objects from

1. INTRODUCTION

Repository In a Box (RIB) is a software package for creating WWW **metadata** repositories. RIB allows the user to enter metadata into a user-friendly java applet that sends the information to an RIB server via HTTP. The information is then stored in an SQL database where it is automatically made available in a fully functional web site (catalog, search page, etc). Repositories that use similar data models can use the XML processing capabilities of RIB to share information via the Internet [1]. While interoperability is one of the most significant features of RIB, it has very limited uses due to the way it is implemented. With the current interoperability of RIB, repositories interoperating with one another can only share **primary class** objects in their catalogs. In real world applications, interoperability is currently possible only as a way to produce a combinational view of multiple repository catalogs. **Interoperation objects** are treated much too differently from **local objects** to become truly useful to repositories that create their interoperations. Also, other RIB features cannot benefit from the “interoperation feature,” leaving this powerful feature isolated and useless. The work presented in this report explores how to improve the interoperability of RIB, how to make interoperation work better with other RIB features, and, it shows how interoperation can be made a real selling point of RIB.

2. METHODOLOGY

This section will discuss three ways in which the interoperability of RIB can be improved. First, the “interoperation feature” is improved, making interoperation objects usable by repositories creating interoperations. Second, interoperation is incorporated with the “join

feature,” making interoperation objects displayable in the “join matrix.” Third, the “import feature” of the current RIB is improved in such a way that importing objects is done as a recursive and interoperation compatible process. Improvements made in all three phases will help achieve the overall goal of improving the interoperability of RIB.

2.1 Improving the “interoperation feature”

Interoperation allows repositories using similar data models to share metadata with each other. The shared metadata objects are automatically linked into the catalog of the repository that creates the interoperation [1].

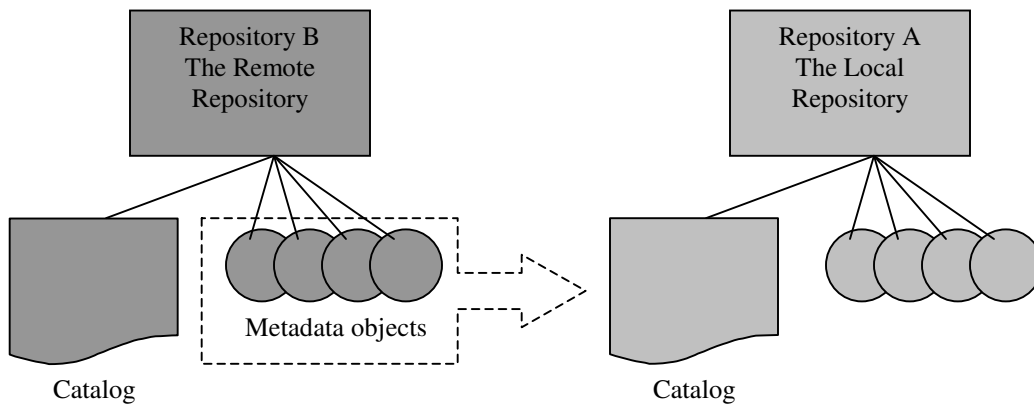


Figure 1: Interoperation between two repositories [1]

As shown in Figure 1, the objects maintained in Repository B (the **remote repository**) are linked into the catalog of Repository A (the **local repository**) as a result of an interoperation created at Repository A. When the user clicks a link in the local repository’s catalog to view an interoperation object, that object is retrieved from the remote repository, not from the local repository. This ensures that the information in the local repository’s catalog is always current, and it also ensures that the repository

responsible for maintaining the object controls its appearance and access permissions. The primary class of the repository that creates the interoperation will dictate the class of linked objects.

In the current implementation, interoperation objects are stored in a dedicated database table called an interoperation table, whose fields are defined very differently from other class tables, as will be shown later. Because of this, interoperation objects can only be viewed from the catalog and cannot contribute to the construction of the local repository. Because local objects have no knowledge of the existence of these interoperation objects, local objects cannot create meaningful **relationships** with them. Therefore, the reusability of these interoperation objects is very limited. In order to improve the “interoperation feature,” the most important step is to make interoperation objects usable by other local objects. Approaches to achieve this goal are explained below.

2.1.1 Storing interoperation objects in class tables together with local objects

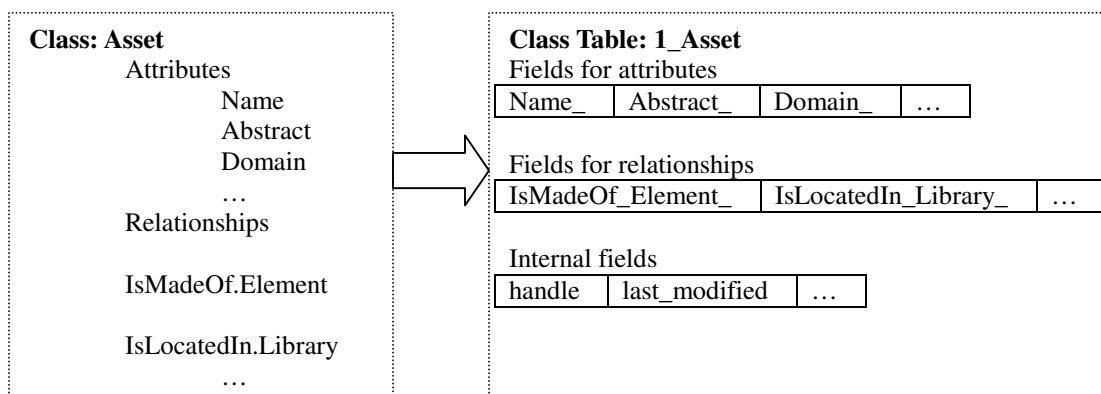


Figure 2: Underlying database table structure of an RIB class

As shown in Figure 2, each RIB class will correspond to an SQL database table. Local objects are stored in their own class tables as individual records. To make interoperation objects more useful, they also need to be stored in these class tables together with other local objects. Because interoperation objects are just “pointers” to objects in remote repositories, they do not have to duplicate the contents of their remote correspondences. They only need to keep the minimum information about the remote objects they point to in order to retrieve them. A “*link*” field is added to each class table to serve this purpose. For a local object, this field will always be “*null*,” but, for an interoperation object, this field will contain the URL of the **remote object** it points to. Considering the fact that more than one interoperation can be created in a repository, an “*owner_handle*” field is needed to identify which interoperation a specific interoperation object belongs to. As with local objects, interoperation objects are also given an object handle. Unlike local objects having positive numbers as object handles, interoperation objects are assigned negative numbers as object handles. This makes an interoperation object recognizable simply by looking at its object handle. In summary, the underlying database table structure of an RIB class is modified, as follows, to accommodate the introduction of interoperation objects:

- a. An internal “*link*” field is added to each class table;
- b. An internal “*owner_handle*” field is added to each class table;
- c. Interoperation objects are given negative numbers as object handles.

2.1.2 Updating interoperations to make interoperation objects consistent with remote objects

When updating an interoperation, information about objects from all classes in the remote repository will be retrieved, and interoperation objects from all common classes in the local repository will be updated accordingly. As illustrated in Figure 3, different cases need to be considered.

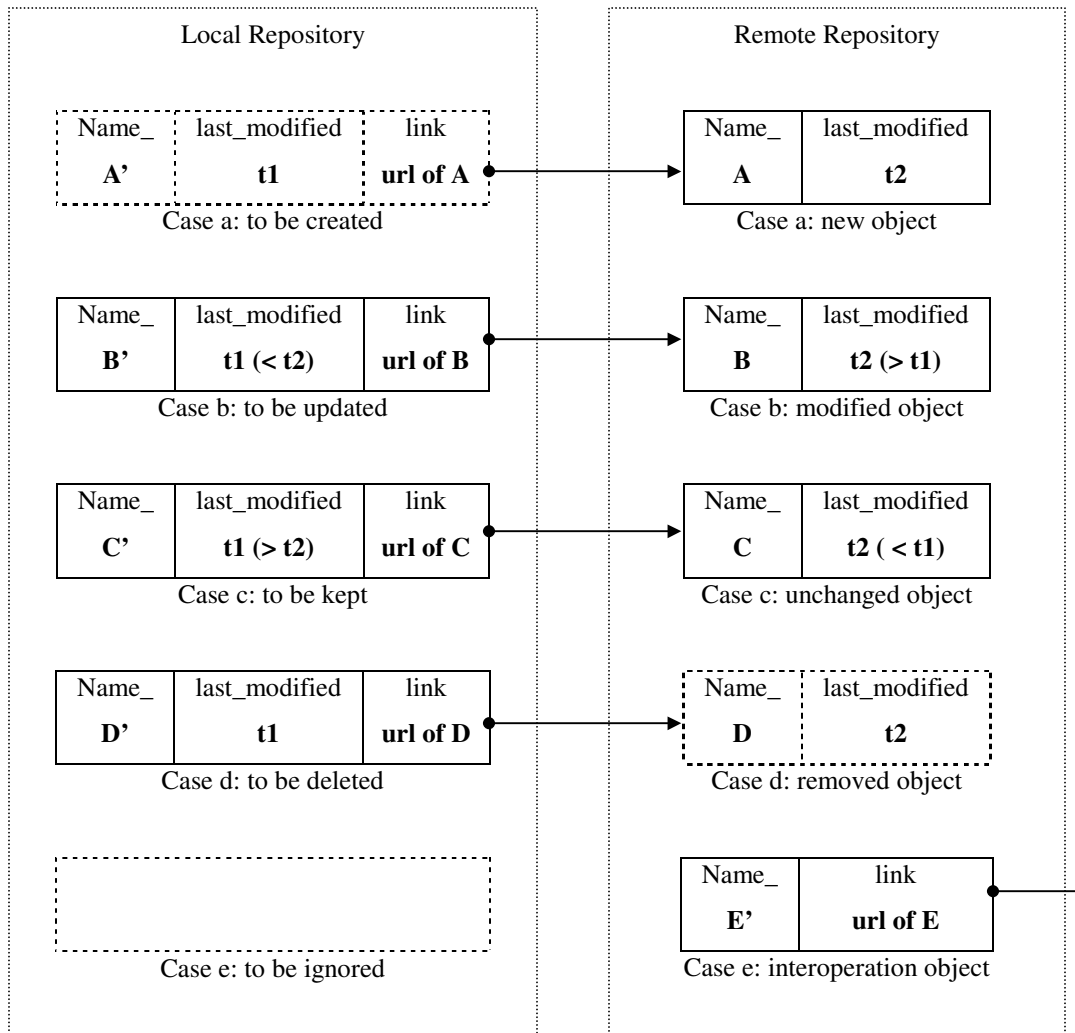


Figure 3: Cases to consider when updating interoperations

Remote object is new

When the remote object is new, an interoperation object will be created in the local repository to point to it. The interoperation object will be assigned a negative object handle that follows a sequence and will copy the “*Name*” **attribute** of the remote object. Its “*link*” field will contain the URL of the remote object and its “*owner_handle*” field will be set to the handle of the interoperation that is being updated.

Remote object is modified

When the remote object is modified, an interoperation object pointing to the remote object already exists in the local repository, and, the remote object has a greater value in its “*last_modified*” field than that of the interoperation object. The “*Name*” attribute of the interoperation object is the only field to be updated specifically, while the “*last_modified*” field of the interoperation object will be reset to the current time automatically by the database system.

Remote object is unchanged

If the value in the “*last_modified*” field of the remote object is less than that of the interoperation object, the remote object is considered unchanged. Nothing has to be done in this situation. The interoperation object that already exists in the local repository will simply be kept unchanged as well.

Remote object is removed

If an object in the remote repository is removed, the interoperation object pointing to it has no reason to exist in the local repository. In this situation, the interoperation object will be deleted from the local repository. When it is clear that information from removed objects cannot be retrieved when contacting the remote repository, how does an interoperation object know the remote object it points to has been removed? Solving this problem requires keeping a list of all the existing interoperation objects that belong to the interoperation currently being updated. Each time a remote object is examined, whether it is new, modified or unchanged, the interoperation object pointing to it will be removed from the list, but not deleted from the repository. After all of the remote objects have been examined, interoperation objects still left in the list will be the ones that need to be deleted.

Remote object is also an interoperation object

Due to the fact that the introduction of “pointer to pointer” makes little sense to the interoperability of RIB, and increases complexity, no interoperation object will be created in the local repository to point to another interoperation object in the remote repository.

When that is the case, the remote interoperation object will just be ignored.

2.1.3 Working with interoperation objects in the administration interface

In the administration interface, interoperation objects will be listed together with local objects. The symbol “*” will denote these objects and make them identifiable. Because interoperation objects are “pointers” to remote objects, they cannot be edited, nor can

they be attached with files. When trying to view an interoperation object from within the administration interface, either in XML format or in HTML format, the actual remote object it points to will be seen. Because interoperation objects are also stored in class tables, and have their own object handles, they can be referenced through their object URLs. This very useful feature enables local objects to create relationships to interoperation objects as if these interoperation objects were local. In a sense, objects created by remote repositories can be “used” by the local repository. This reinforces the concept of “metadata reuse,” which is the key purpose of providing various interoperation functions in RIB.

2.1.4 Viewing interoperation objects in the catalog

Interoperation objects will not appear in the catalog by themselves. An interoperation object will only show up in the catalog as the value of a relationship field of a local object. When users click on the hyperlink of an interoperation object, they will not be able to view the interoperation object itself. Instead, they will be taken directly to the remote object it points to. By doing this, users of the catalog will not notice any difference between a local object and a remote object, nor will they notice that they are jumping back and forth between different repositories. When this is implemented, seamless connections among repositories are achieved.

2.1.5 Deleting interoperation objects

When deleting an interoperation, all of the interoperation objects that were created by the interoperation will also be deleted. Since local objects may have relationships to

interoperation objects, when an interoperation object gets removed, all relationships to it should also be removed. These relationships can be removed by examining the relationship fields of each local object in every class.

2.2 Improving the “join feature”

“Join” is an advanced feature of RIB that allows users to query the repository data in a special way. A class that is the intersection of two other classes can be displayed in a matrix [1].

For example, suppose there are three classes in the repository: *Asset*, *Machine*, and *Deployment*. Also, suppose the *Deployment* class has a relationship to the *Asset* class called “*IsDeploymentOf*” and another to the *Machine* class called “*IsDeployedOn*.” Here, the *Deployment* class serves as the intersection class because it has relationships to each of the other two classes. The arrangement is illustrated below in Figure 4:

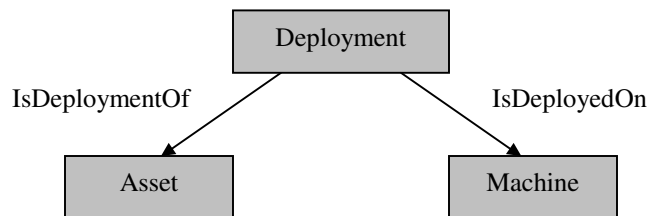


Figure 4: Relationships among classes [1]

Next, suppose several *Asset*, *Deployment*, and *Machine* objects are created, with each *Deployment* object having both an “*IsDeploymentOf*” relationship to an *Asset* object and an “*IsDeployedOn*” relationship to a *Machine* object. Also, suppose the *Deployment* class

has an attribute called “*Status*,” which can be given the values “*Successful*,” “*Incomplete*” or “*Aborted*.”

Now create a custom “join” and specify the “join” parameters as described below:

Intersection class: *Deployment*

Intersection relationship 1: *IsDeploymentOf.Asset*

Intersection relationship 2: *IsDeployedOn.Machine*

Interior attribute: *Status*

At last, a “join matrix” will be produced as shown in Figure 5:

	Machine X	Machine Y	Machine Z
Asset A	Successful	Incomplete	Aborted
Asset B	Successful	Successful	Incomplete

Figure 5: The “join matrix” [1]

The current implementation of “join” does not consider interoperation objects. From the user’s point of view, it is very natural to expect that interoperation objects will also be able to show up in the final “join matrix,” given that the remote objects they point to can fit well into the matrix.

2.2.1 Design concerns: Constant updating versus instant retrieving

For an object of the intersection class to be able to show up in the “join matrix”, it must contain valid values in the three fields specified as “join” parameters: the two intersection relationship fields and the interior attribute field. Conversely, the object will not show up if any of these fields contain the “*null*” value. Interoperation objects will not keep these values in the local repository because they are “pointers” to remote objects. Therefore, these values need to be retrieved from the remote objects they point to.

There are two different approaches to retrieve the aforementioned values. The first approach is to retrieve the values from remote repositories and store them in interoperation objects while updating an interoperation. Every time an interoperation gets updated, the value of each field in each interoperation object needs to be updated accordingly. This approach is called **constant updating**. When constant updating is implemented, interoperation objects are no longer simple “pointers” because they have to store remote content locally. Updating an interoperation in this manner can be very expensive because updating every single interoperation object requires retrieving the content of a remote object, which will result in increased HTTP requests over the Internet. The main concern with this approach is that a lot of information gets wasted due to the fact that the values will only be consumed by “join” operations, which are not carried out very often.

An alternative approach to retrieve values from remote objects is called **instant updating**. Instant updating retrieves values when they are needed, that is, when the user initiates a

“join” operation. Although the same set of values may be retrieved multiple times in a sequence of “join” operations of the same kind (an unlikely scenario), the information that is retrieved each time is reduced to a minimum. This is the method that will be discussed in this report.

2.2.2 Retrieving the content of the remote object

For any interoperation object, the content of the remote object it points to can be retrieved by sending an HTTP request to the object URL stored in the “*link*” field of the interoperation object. This content can then be obtained as an XML document. Parsing this XML document can extract all values needed to carry out a “join” operation on this interoperation object.

2.2.3 Replacing retrieved values to ensure local relationships

As mentioned in section 2.2.1, values that are to be extracted from the remote object’s XML are values of the two intersection relationship fields and the interior attribute field. The value of the attribute field can be copied into the interoperation object without any problem. However, relationship field values cannot be copied into the interoperation object directly. As defined in RIB, the value of a relationship field is the URL of another object. Copying these URLs directly from a remote object to its interoperation object will result in cross-repository object references, which contradicts the RIB design conventions. The convention requires that all relationships be local references, and cross-repository references are only allowed to appear in the “*link*” fields of interoperation objects. This problem can be easily solved if interoperation objects are made to be useful. After the

relationship field values of the remote object are extracted, “pointers” to these values will be placed in the corresponding fields of the interoperation object instead of the values themselves. Searches in the local class tables will try to resolve these “pointers,” which are nothing more than other interoperation objects. Careful examinations need to be made in consideration of different situations as illustrated in Figure 6 and 7:

- a. Extracted relationship values are URLs of local objects in the remote repository

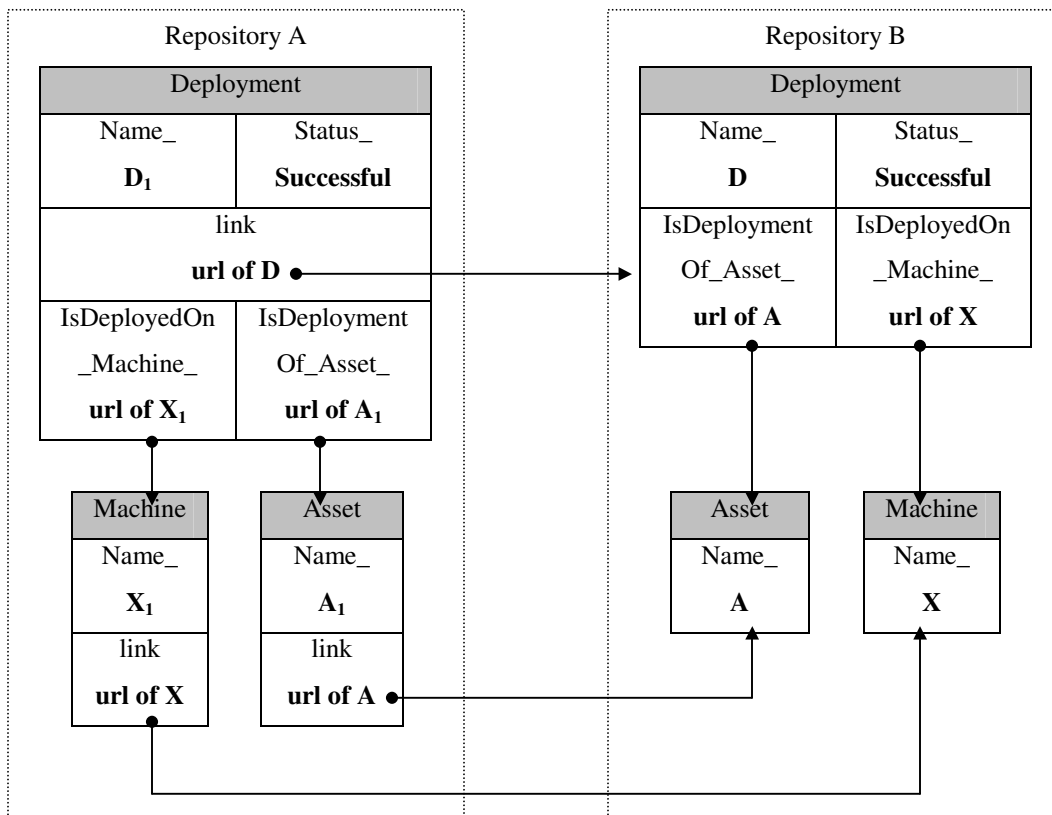


Figure 6: Situation A

In this situation, the URLs of objects A and X will be extracted from object D’s XML for the “*IsDeploymentOf*” and “*IsDeployedOn*” relationship fields. As shown in Figure 6, interoperation objects A₁ and X₁ are “pointers” to remote objects A and X. Since A and X are local objects in Repository B, and Repository A has created an interoperation with

Repository B, “pointers” A_1 and X_1 are guaranteed to be found in Repository A.

Therefore, the URLs of objects A_1 and X_1 will be placed into the relationship fields

“*IsDeploymentOf*” and “*IsDeployedOn*” of object D_1 .

- b. Extracted relationship values are URLs of interoperation objects in the remote repository

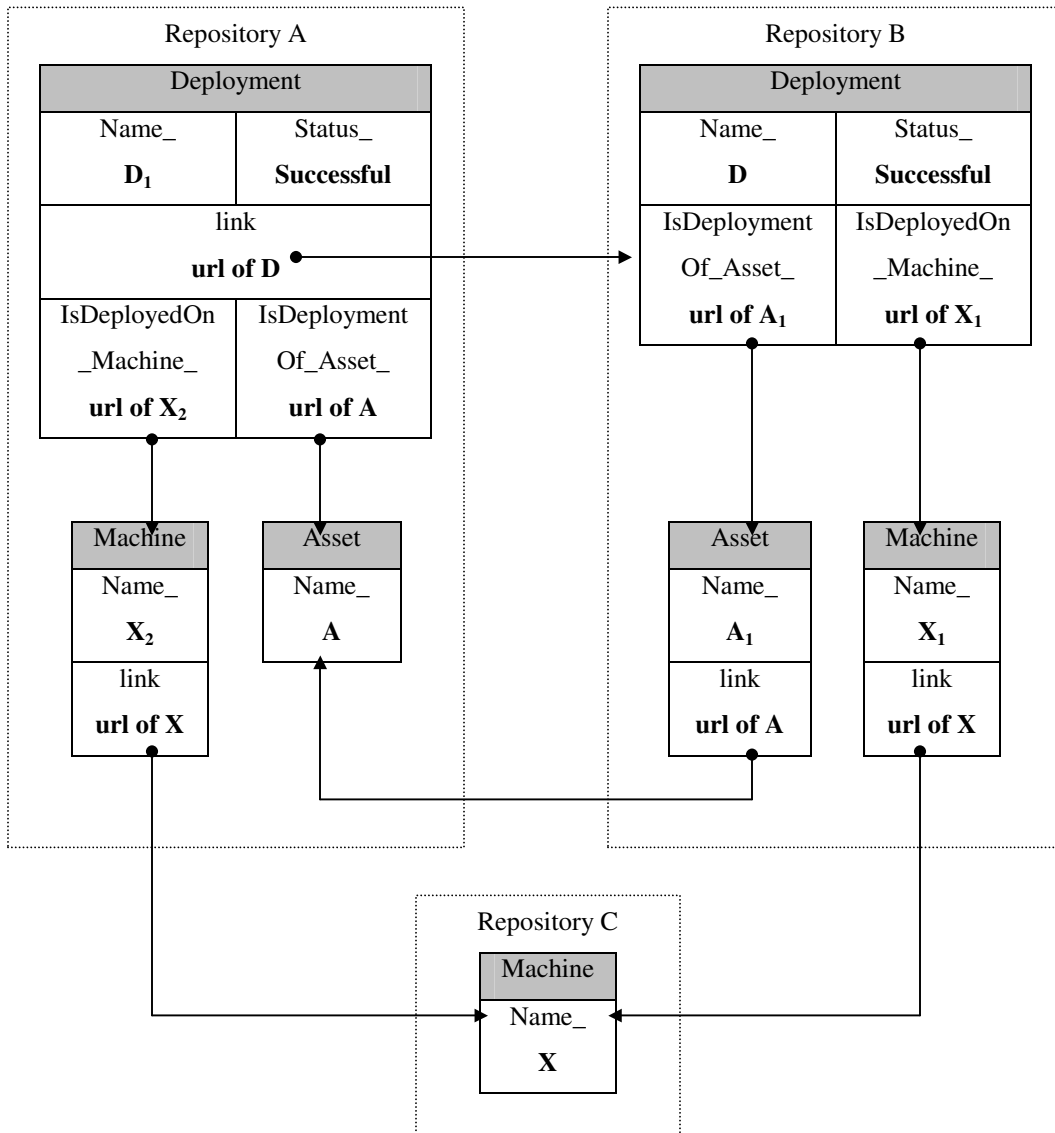


Figure 7: Situation B

In this situation, the URLs of object A_1 and X_1 will be extracted from object D 's XML for the “*IsDeploymentOf*” and “*IsDeployedOn*” relationship fields. Since A_1 and X_1 are now interoperation objects in Repository B and, as mentioned in section 2.1.2e, “pointer to pointer” is not introduced in RIB, interoperation objects that are “pointers” to A_1 and X_1 will not exist in Repository A. So, in addition to retrieving the content of object D , the contents of objects A_1 and X_1 have to be retrieved in order to find out what remote objects they point to. In the case of X_1 , the remote object it points to is X in Repository C. Therefore, interoperation object X_2 , which is a “pointer” to X , can be looked up in Repository A. Notice here that X_2 can be found only if Repository A is also interoperating with Repository C, otherwise, X_2 cannot be found. The remote object that A_1 points to is A , which is located inside Repository A. Therefore, no interoperation object that is a “pointer” to A should be looked up in Repository A, and, object A itself will be used. At last, the URLs of objects A and X_2 will be placed into the relationship fields “*IsDeploymentOf*” and “*IsDeployedOn*” of object D_1 .

2.2.4 Clearing temporary values in interoperation objects

Due to the fact that interoperation objects are simply “pointers” to remote objects, they are not supposed to store remote content locally. All values retrieved for an interoperation object for “join” purposes are temporary values that need to be deleted at the end of the “join” operation in order not to cause any problems to other operations. If these temporary values remain there, they may confuse operations that are not expecting any values in the relationship fields of an interoperation object.

2.3 Improving the “import feature”

Importing an object provides the local repository with a snapshot of an object from the remote repository. Unlike interoperating with another repository, which ensures that the information appearing in the local repository always stays current with the metadata in the remote repository, importing an object captures information maintained by the remote repository in static-copy form. Thus the changes made by the remote repository after the object has already been imported will not be reflected in the local repository [1].

Importing is not done as a recursive process in the current RIB implementation. If a remote object has relationships with other objects, those objects will not be imported as well, and the **imported object** will still refer to them in its relationship fields, leaving cross-repository references. When a local object with cross-repository references is displayed in the catalog, the references will be shown as “external information,” which can be confusing to many users. As mentioned in section 2.2.3, to be consistent with RIB design conventions, recursive importing has to be implemented in order to get rid of cross-repository references. Elimination of cross-repository references in imported objects ensures that only interoperations can introduce connections to remote repositories. Therefore it reinforces the idea of interoperation from another aspect.

Importing should also be interoperation compatible, that is, interoperation objects should be recognized and handled properly without causing problems in the importing process.

2.3.1 Design concerns: Depth-first versus width-first

Importing objects recursively means importing the specified object as well as any other objects referenced through its relationships. All the objects that are to be imported will form a tree-like structure, which is called an importing tree in this report. The initial object to be imported is the root node, and all other objects directly or indirectly referenced by it are child nodes. Relationships that occur in between objects are edges in between nodes. This is not exactly a tree because objects related to one another can form loops, which must be resolved to avoid problems. Objects of the importing tree can be traversed either in depth-first order or in width-first order. From an implementation standpoint, both of them are practical methods, and each one is superior to the other in certain cases. Considering the facts that Perl is not a well structured programming language and that Perl code is usually difficult to read, introducing recursive function will make the code even harder to produce and understand. Therefore, the width-first approach is adopted, in which the use of recursive calls can be avoided and a FIFO queue will suffice.

2.3.2 Importing objects using width-first traversing of the importing tree

Before the initial object is imported, its URL will first be placed in a FIFO queue, which will store the URLs of all the objects to be imported. URL of the next object to be imported will be taken from the head of the queue, as is what a FIFO queue requires. The actual content of the remote object identified by this URL will be retrieved from the remote repository as an XML document through an HTTP request. A local copy of this remote object can be created directly from this XML document. When examining the

XML document, all relationship fields need to be identified. All references to other objects will be appended to the FIFO queue in order to import those objects later. Object URLs will be obtained from the FIFO queue repeatedly, until there are no more left. This process is identical to traversing the importing tree in width-first order and it guarantees the ability to import all objects involved.

The following identifies three different situations that need to be considered when importing an object, and explains how to deal with them. These situations are also illustrated in Figure 9 for clarity.

Object to import is a local object in the remote repository

In this case, a regular local object needs to be created with all its content copied from the remote object. Passing the obtained XML document of the remote object to the “object creator” module will create an identical copy of the remote object in the local repository. Importing object B and C from Repository B in Figure 9 shows this case.

Object to import is an interoperation object in the remote repository

In this case, a local interoperation object of the same name needs to be created with its “*link*” field copied from the remote interoperation object. Extracting only the “*Name*” and the “*link*” attributes from the obtained XML document will be enough to create such a simplified copy. Because this interoperation object is not created from any interoperation established by the local repository, it is a special interoperation object with the

“*own_handle*” field set to 0. Importing object D₁ from Repository B in Figure 9 shows this case.

Object to import is an object already imported

To avoid importing the same objects again and again requires resolving loops in the importing tree. A hash table is built for this purpose. For each entry in this hash table, the “*key*” is the URL of the remote object, and the “*value*” is the URL of the imported object. Each time a remote object is successfully imported and a local object is created for it, the (*key*, *value*) pair is added to the hash table. As in Figure 9, after object A, B, C have been imported, the hash table will look like:

Key	Value
URL of object A	URL of object A ₁
URL of object B	URL of object B ₁
URL of object C	URL of object C ₁

Figure 8: The hash table

This hash table can then be used to avoid looping in the importing process. Each time an object URL is obtained from the FIFO queue, the hash table will also be examined to find out whether there’s an entry keyed by this URL or not. If there is, which means the remote object identified by this URL has already been imported, the next object URL will be fetched from the queue; if there isn’t, the remote object identified by this URL will be

imported as described in section 2.3.2. Given the hash table in Figure 8, object A will not be imported for the second time although object B refers to it as shown in Figure 9.

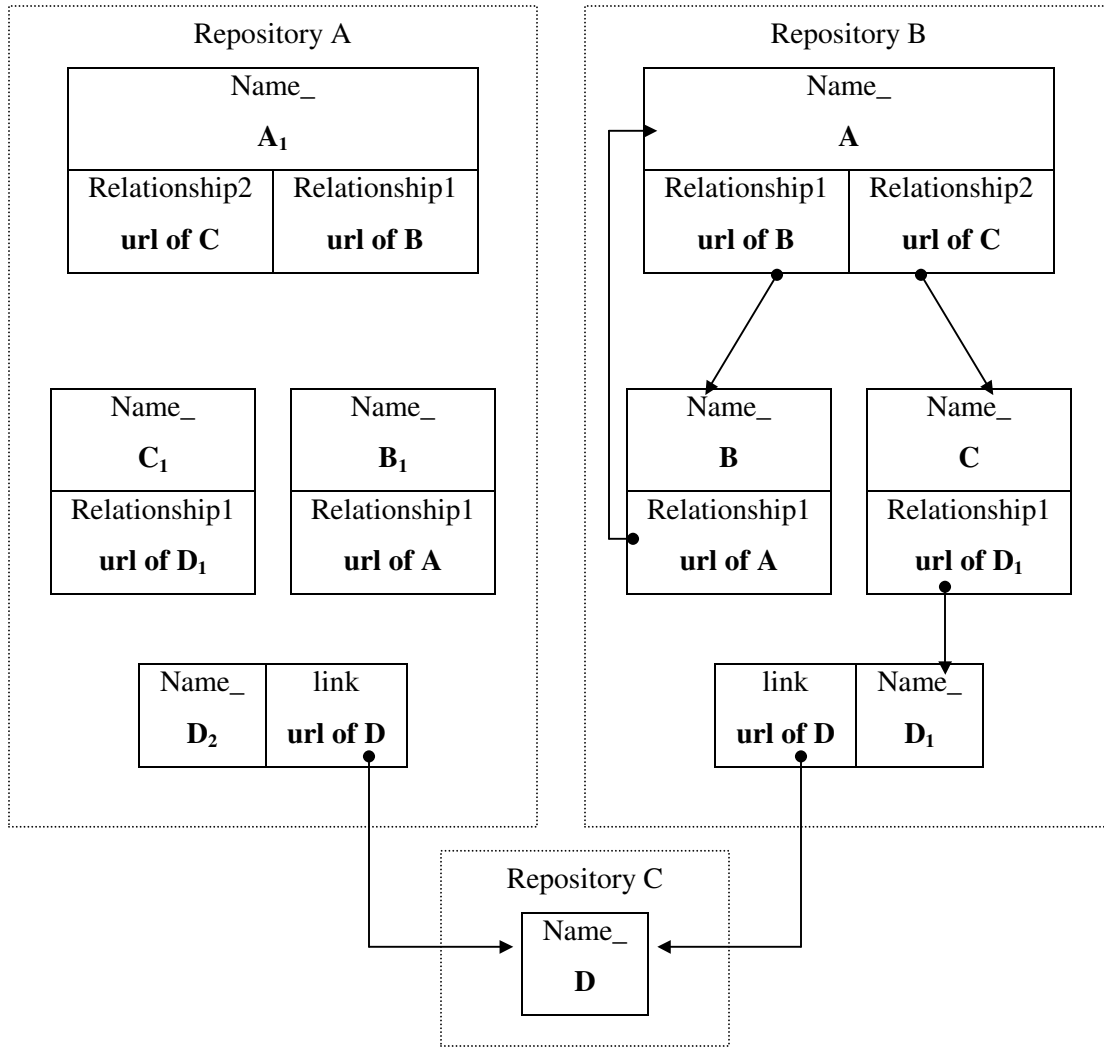


Figure 9: Illustration of 3 different situations

2.3.3 Updating relationship fields to reflect local relationships

Even though all objects in the importing tree have been imported, the work is not done yet. Since imported objects are created directly from the XML documents of their remote

correspondences, the values of their relationship fields are still the original values copied from the remote objects. Cross-repository references have not been eliminated. The hash table described in section 2.3.2c can also be used to solve this problem. For every registered (*key*, *value*) pair in the hash table, each imported object in every class will be examined, and the “*key*” will be replaced with the “*value*” for every relationship field where the “*key*” appears. After this step is completed, imported objects in Repository A will have the correct local relationships, as illustrated in Figure 10:

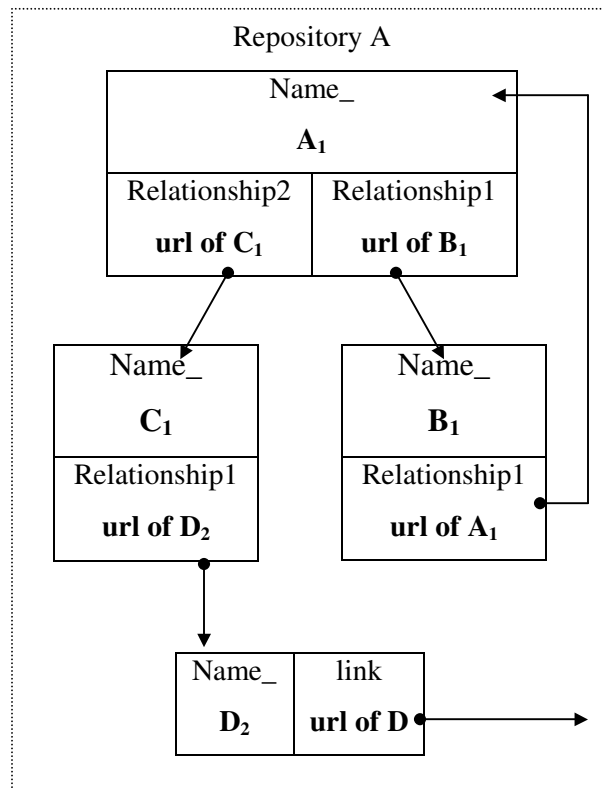


Figure 10: Updating relationship fields

3. CONCLUSIONS

Improving the “interoperation feature” makes interoperation objects usable by local objects and the central idea of metadata reuse is reinforced. Incorporating interoperation with the “join feature” demonstrates possible ways to pack the power of interoperation into other RIB operations. Improving the “import feature” helps to eliminate unexpected cross-repository references, making the idea of interoperation clearer. Overall, the interoperability of the current RIB is greatly improved, bringing it broader applications.

4. APPLICATIONS

The most direct application of RIB’s improved interoperability is to maintain the same repository by different organizations. Organizations can create and maintain metadata objects in specialized areas, and then they can create interoperations with one another. This way, the work of different organizations can be assembled together. Metadata objects created by other organizations can be linked into an organization’s local repository and can be referenced by its local objects, avoiding recreation of these objects. Work results of another organization, like the deployment of software on machines, can be immediately accessible in an organization’s local “join matrix” once the proper interoperations are created.

5. FUTURE WORK

Since RIB is a demand-driven project, development and improvement cannot cease as long as there are new user demands. The improved interoperability only provides a general framework demonstrating how interoperation can be effectively implemented and

how further extensions can be carried out under this framework. It is also very likely that users will have additional requirements for RIB's interoperability. So, future work on the interoperability of RIB, for the most part, will consist of creating user-customized features to meet specific user needs, solving specific problems users have, and making the interoperation among repositories more powerful and more useful.

ACKNOWLEDGEMENT

I would like to give special thanks to my advisor, Dr. Shirley Moore, for helping me find the RIB project and encouraging me along the process. I would also like to thank Dr. Jack Dongarra and Dr. Michael Berry for advising me on this project. Thanks to Don Fike for answering all my questions on the system issues of RIB, to Scott Wells for giving me the background of RIB and to Nathan Garner for shipping me the latest Java applet source code.

REFERENCES

[1] Repository in a Box Homepage, *<http://www.nhse.org/RIB>*.