Extending the Remote Exchange framework to suit instance-based interoperability

Mark Vermeer

December 5, 1996

Abstract

In this document, I discuss the Remote Exchange project, an import-export-based sharing mechanism implementing a database federation. Based on my own research, I propose some extensions to this mechanism and indicate how they might be implemented.

1 Introduction

During my stay at the University of Southern California (USC), I have studied the Remote Exchange (RE) project that has been carried out at USC during the last few years. This document serves to account for the work that I have done during this stay. It describes the RE project and its correspondences with my own work. Based on ideas I developed, I propose some extensions to RE and describe how they could be implemented.

This document is organised as follows. I first give a description of the principal features of the RE system in Section 2. In Section 3, I then describe the idea of instance-based interoperation, which stems from my earlier work. Section 4 describes how this paradigm relates to the current RE functionality. Section 5 contains some proposals for extensions of RE in the light of these ideas.

2 An overview of Remote Exchange

In this section, I will describe the main features of RE as I see it. My sources are publications about the project such as [1,2,3,4]
2.1 General principles

RE is a system based on the federated approach to database interoperability proposed in [5]. That is, unlike many other approaches, RE does not define some integrated interoperable level, spanning all or a number of component databases. Instead, RE allows a component to import data exported by other components into the local database, stressing the autonomy of components. Thus each component is enriched with imported data in a transparent way, without directly affecting the local user interfaces or query processing mechanisms.

An important consequence of this is that instead of defining an integrated schema that subsumes all local schemata, each component transforms remote data to fit its local context. This may involve extensions to the local schema, but conflicts are always resolved in favour of the local representation. Also, a component will only import remote information that is deemed relevant.

The whole project consists of three tiers:

1. The discovery and identification of remote information relevant to a given component.
2. The resolution of semantic heterogeneity between this component and the exporter of relevant information.
3. The actual mechanism that enables these components to share information.

2.2 Discovery and Semantic Heterogeneity Resolution

To import objects from a remote source in a meaningful way, information on the meaning of remote objects is needed, so that an importing component may determine whether information provided by a remote data source is relevant to him. Moreover, there may exist discrepancies in the way similar information is represented in the importing resp. exporting component. Hence on top of the actual sharing mechanism, we need functionality to discover relevant remote information and to resolve possible semantic heterogeneity. This functionality is part of the first two tiers of RE listed above. These are discussed in this subsection.

The main idea underlying the RE approach to these issues is that one should not reason about the meaning and resemblance of heterogeneous objects in terms of their structural representation in the individual components (i.e. schematic information), but use semantic knowledge about these
objects instead. In particular, a *Semantic Dictionary* (SD) is maintained, that serves as a federated knowledge base of the application domain covered by the federation components. This SD is dynamic in that it evolves with the registration of new exported information. The SD consists of *concept hierarchies*, where a concept is defined as a collection of similar types.

The actual process of discovery and unification in RE is governed by the *Sharing Advisor* (SA), a component that manages knowledge about type objects that components export. Note that the SA is primarily concerned with type-level sharing. Four SA services are distinguished:

1. *Registration*

When registering new type objects to be exported into the SD, a logical connection is established between the exported information and the existing concepts in the SD. The determination of the relationships between the newly registered type objects and the concepts in the current hierarchy is done using user interaction, supported by a clustering algorithm based on *inter-concept dissimilarity* and *intra-concept similarity* of properties to determine the meaning of a registered type object.

2. *Discovery*

The discovery service takes user (component) characteristics into account when determining relevant remote information. That is, the following three types of requests are supported:

(a) *Similar concepts.* This is a request for types that are in the same portion of the concept hierarchy as a given local type.

(b) *Complementary information.* This is a request for a type that has additional information (i.e. different properties) for a given concept to which a certain local type belongs.

(c) *Related information.* This is a request for a type that shares some of the properties of the concept to which a given local type belongs. This is useful for example for the sharing of behaviour between the local and remote type.
3. **Semantic Heterogeneity Resolution**

The basic problem addressed by the semantic heterogeneity mechanism is the determination of the relationship between a local and a remote (type) object. To this end, *structural* knowledge about type objects is represented by *meta-functions*, which are used to supply meta-data about remote type objects, such as their functions, instances and sub-types.

Moreover, a *Local Lexicon* (LL) represents the *semantic* relationships between local types and globally understood concepts from the SD. Possible relationships include: identity (i.e. same concept, same representation), equivalence (same concept, different representation), compatibility, kind of, association (e.g. synonyms; typically used in the same context), collection of, has (a property that is inherent to the type), and descriptive feature of. Hence the LL represents the semantics of shared terms in a more expressive and complete manner than in the conceptual schema.

Thus, in RE the “meaning” of concepts is approximated using a combination of the SD, which contains partial information on inter-component concept relationships, and the LL, which is an attempt to describe the precise meaning of local types in terms of globally understood concepts.

4. **Unification**

When folding relevant remote information into the local database, semantic discrepancies between the local and remote component must be dealt with. For example, the values of primitive stored functions may be *converted* to some common domain. Moreover, functions and/or types may be *renamed* to deal with homonyms and synonyms.

When the semantic relationship between a remote type and a local type has been determined, this type can be added to the local schema. The way in which this is done depends on the relationship involved. As we touch on the actual sharing mechanism here, this is discussed in the next subsection

2.3 **The sharing mechanism**

Three sorts of objects to be shared are distinguished:
1. Instance objects.

The import of remote instance objects can be seen as the integration of remote information into the local context. Locally, surrogate objects are created to represent the imported objects. The functions associated with the local type under which the remote object is imported, are overridden to do an RPC call of the appropriate remote function associated with the original remote object (this is done using multiple typing for surrogate objects, i.e. surrogate objects are both of a 'surrogate remote' type and of the type they are imported into, and corresponding dynamic binding of overloaded functions).

2. Type objects.

The import of remote type objects can be seen as the embedding of a remote context into the local context. A remote type is always imported as a subtype of a local type, and subtypes of the imported type are automatically imported as well. Typically, importing a remote type involves the sharing of both the metadata objects and the remote instances that form its extension. Subsequently, additional local instances may be added to these remote types.

When importing a single remote type $R$ into a local component, two types of relationships between $R$ and a local type $L$ are distinguished:

(a) $R$ is semantically equivalent to $L$.

   Now $R$ is locally created as a subtype of $L$, and all its functions are created for both $R$ and $L$.

(b) $R$ is related to $L$.

   Now the well-known generalisation technique is used: both $R$ and a new supertype $S$ are created, where $S$ is a supertype of both $R$ and $L$. $S$ contains the common functions of $R$ and $L$.

If there is no such relationship, $R$ is simply imported as the root of a new hierarchy.

If multiple, inter-related remote types are imported, first the type objects are imported as sketched above, followed by the functions that relate them. Finally, in the presence of representation differences of relationships between equivalent local and remote types, a surrogate for the local relationship representation is created for each pair of related imported objects.

These objects can be either stored functions (often called *attributes* in other data models), or computed functions (which correspond to *object retrieval methods*). By sharing behaviour, a component may access remote services for its objects that are not supported locally. Sharing of behaviour can be done in either of the two following ways. *Implicit* behaviour sharing is performed when importing remote instances; its associated functions are shared automatically. *Explicit* behaviour sharing is done by first importing remote function objects, and then associating them with (local) types. A *callback mechanism* is supplied for shared remote functions to access the state of local objects they are executed on.

Hence, in contrast to most existing approaches, the unit of sharing between components is not necessarily a *collection* of objects (or, in particular, the complete extension of a certain remote type); instance objects are an appropriate unit of sharing here, and even behavioural objects (functions).

**A behaviour sharing taxonomy**  An important feature of RE is the decoupling of location of data and behaviour. Hence we may execute:

1. Local functions on local objects
   
   This is the base case, regardless of whether the function is stored or computed.

2. Local functions on remote objects
   
   This has the effect of giving local state (stored functions) or local behaviour (computed functions) to a remote object. Thus these objects are adapted to the local context.

3. Remote functions on local objects
   
   This only makes sense if the function is computed; in that case, the remote database is providing a service to local objects that cannot be supplied locally. This may be seen as the purest form of behaviour sharing.

4. Remote functions on remote objects
   
   This is the basis of the instance sharing mechanism; as pointed out above, behaviour sharing is implicit in this case.
3 Instance-based database interoperation

It may be observed that there exists a certain gap between the two principal components of RE: the sharing advisor and the actual sharing mechanism. Whereas the focus of the sharing advisor is on detecting relationships between and the unification of local and remote types, the actual sharing mechanism is mostly concerned with the sharing of instances and behaviour. A possible way to bridge this gap is by applying the principle of instance-based database interoperation. In this section, we present the main ideas behind this paradigm, as presented in [6,7].

3.1 Objects and classifications

Observe that the notion of a schema is essentially a classification of the set of real-world objects appearing in a certain application domain. Objects that are considered similar by the schema designer are grouped into entity types (classes) and described using a common set of properties (attributes). If the set of real-world objects grouped by a certain entity type \( E \) is a subset of the set of real-world objects grouped by another entity type \( E' \), \( E \) is called a subtype of \( E' \).

We note that the classification of real-world objects into entity types is inherently subjective; different designers will do it in different ways, depending on their way of looking at the application domain. Moreover, due to differences in context, it is often very hard to determine the exact relationship between a locally defined entity type and a remotely defined entity type. Take as a simple example the entity type Person defined in the database of the CS-department of university X, and the entity type Student defined in a database for the whole university X. We would argue that the apparent ISA-relationship between these types does not hold here, since the more general type is defined in a more specific context. Also consider entity types like Federated DBS versus MultiDBS or even AmericanCars versus BeautifulThings.

In short, our idea is to avoid having to define relationships between classes as much as possible. Instead, we try and define relationships between instance objects. We then apply both the local and the remote classification to an appropriately merged “global” set of objects.
3.2 Specifying interoperability

Our approach requires a designer to specify conditions under which a certain relationship between a remote object $O'$ and a local object $O$ or class $C$ holds. The relationships we distinguish are:

- **Identity.** $O$ and $O'$ represent the same real-world object. This is represented as $\text{Eq}(O', O)$.

- **Strict similarity.** $O'$ would locally be classified under $C$. This is represented as $\text{Sim}(O', C)$.

- **Approximate similarity.** Locally $C \cup \{O'\}$ would be considered a meaningful more general class $C''$. This is represented as $\text{Sim}(O', C, C'')$.

- **Descriptivity.** Locally $O'$ is considered a set of values $S$ describing an object $O''$ which is identical to a local object $O$ or similar to a local class $C$. This is represented as $\text{Eq}(O', O.S)$ or $\text{Sim}(O', C.S)$.

- **Constituency.** Locally $O'$ is seen as a constituent of $O$; $O$ and $O'$ describe different levels of aggregation. This is represented as $\text{Aggr}(O, O')$.

We require the specification of object comparison rules (ocr’s) of the form $\rho \leftarrow \Psi$, where $\rho$ is any of the relationships listed above, and $\Psi$ is a conjunction of first-order logic predicates, which might involve additional information such as correspondence tables etc.

Moreover, property equivalence assertions (proeqs) may be formulated for each pair of classes between which ocr’s have been specified, specifying to what extent the descriptions provided by $DB$ and $DB'$ overlap. This is just what is done in regular schema integration. These assertions are of the form $\text{proeq}(C.p, C'.p', cf, cf', df)$, where:

- $p, p'$ are basic or derived local and remote properties, respectively,

- $cf, cf'$ are conversion functions mapping the domains of $p$ and $p'$ to a common domain $D$, and

- $df : D \times D \rightarrow D$ is a decision function which determines a global value for the property given possibly different local and remote values. We require that for each decision function $df$, $\forall a \in D \{df(a, a) = a\}$. In our view, functions such as $\text{sum}$ used e.g. in [8] define derived global properties rather than determining values for equivalent local and remote properties.
3.3 Example

Figure 1 depicts the schemata of the literature databases maintained by a researcher A and a researcher B, respectively. Note that the databases contain similar information, but there appears to be some discrepancy in the classifications used. For example, the term “Publication” as used by researcher A may very well be closer to what B calls a “Research paper” than a “Publ”. Moreover, A distinguishes subclasses based on the type of publication forum, while B considers the organisation responsible for the publication.

Suppose researcher B would like to import some information from researcher A into his own database. He might then define the following object comparison rules:

\[
\text{Eq}(O : \text{Publ}, O' : \text{Publication}) \leftarrow O.\text{title} = O'.\text{title} \land \exists a \in O.\text{authors}, a' \in O'.\text{authors} | a.\text{name} = a'.\text{name} \\
\text{Sim}(O' : \text{Conference}, O : \text{IEEE}) \leftarrow \text{contains}(O'.\text{name}, \text{“IEEE”}) \\
\text{Sim}(O' : \text{Journal}, O : \text{IEEE}) \leftarrow \text{contains}(O'.\text{name}, \text{“IEEE”}) \\
\text{Sim}(O' : \text{Conference}, O : \text{ACM}) \leftarrow \text{contains}(O'.\text{name}, \text{“ACM”}) \\
\text{Sim}(O' : \text{Journal}, O : \text{ACM}) \leftarrow \text{contains}(O'.\text{name}, \text{“ACM”}) \\
\text{Eq}(O' : \text{Publication}.\text{publisher}, O : \text{Publisher}) \leftarrow O'.\text{publisher} = O.\text{name} \\
\text{Sim}(O' : \text{Publication}.\text{publisher}, O : \text{Publisher})
\]
Aggr(\(O^\prime :\text{Chapter}, O :\text{Book}\)) \leftarrow O'.\text{booktitle}=O.\text{title}
Eq(\(O :\text{Auth}, O^\prime :\text{Author}\)) \leftarrow O.\text{name}=O'.\text{name}

Note that the uncertainty about the exact relationship between \text{Publ} and \text{Publication} does not prevent us from defining identity rules between their instances. Strict similarity is more conveniently defined lower in the hierarchy, where meaning becomes more obvious, however.

Function names and domains have been kept similar, leaving only obvious property equivalences, except for the following:

\text{propeq}(\text{Publication.authors, Publ.authors, id, id, union})

4 Instance-based interoperation and RE

The sharing mechanism supplied by RE seems to be an appropriate platform to demonstrate the use of instance-based interoperation in the context of an import/export-based database interoperation architecture. Object comparison rules can be used as a specification layer on top of the sharing mechanism, providing conditions under which different forms of sharing are applied. Furthermore, the functionality provided by the sharing advisor may be used to select candidate local and remote classes between which object comparison rules are to be specified. Thus, specifications consisting of object comparison rules and property equivalence assertions could be used as an interface between these RE components.

4.1 Instance-based interoperation in an import/export paradigm

Adapting the idea of instance-based interoperation to the import/export paradigm, with its stress on local autonomy, the following principles suggest themselves:

- Rely on instance sharing as much as possible.
- Where possible, adhere to the local classification for the mixed set of native and imported instances thus obtained.

In terms of our example of Figure 1, researcher B would import researcher A’s publications, but classify them using his local types where
possible. B apparently does not care whether a publication is a conference paper or a journal paper. A’s objects should be classified according to B’s interests: whether they are published in ACM or IEEE fora.

- Use remote types as surrogate types; import remote types into the local hierarchy only to hold imported instances that cannot be assigned to any local type.

- Use behaviour sharing to access additional functions that the remote types supply for the shared instances.

Note that these principles partly replace the current suggestions for type unification supplied by the Sharing Advisor. To implement these ideas, we must extend the current RE sharing mechanism with some additional functionality. This is discussed in the next section.

4.2 Object relationships and surrogates

Although the idea of controlling the import of remote instances through the definition of object comparison rules seems an attractive one, there is a discrepancy between the notion of conditions for object relationships and the surrogate mechanism that implements instance sharing in RE. The surrogate mechanism is static in the sense that it represents the extension of remote types at a certain point in time. The mechanism does not easily reflect changes to the extension of a remote type. For example, the remote object \( o \) represented by a surrogate \( so \) in the local database might be deleted from the remote database. Although in principle this should result in a corresponding deletion of \( so \) from the local database, such an ‘automatic’ deletion violates local autonomy. Thus autonomy of components causes the instance sharing mechanism to be static.

In the presence of conditions that govern the import of remote instances, as in object comparison rules, this problem is expanded to changes in the state of the original, remote database. For example, an update to a remote object \( o' \) may invalidate the condition \( \psi(o') \) under which \( o' \) was imported into the local database as the surrogate \( so \). Alternatively, such an update may lead to this condition to be true on objects that did not qualify previously. These problems are known as the view maintenance problem; in essence the surrogate mechanism implements a materialised view. In view of the autonomy requirement in RE, I here assume that remote changes are not to be immediately reflected in the local database.
4.3 Object comparison rules and the Sharing Advisor

As I propose to let object comparison rules to play the role of an interface between the Sharing Advisor and the sharing mechanism, the responses of the SA to discovery requests are to be defined as follows:

1. In response to a request for similar information, the SA suggests remote types for the definition of strict similarity rules, along with suggested property equivalences. Such rules lead to the importation of additional instances to the local types.

2. In response to a request for complementary information, the SA suggests remote types on which to define identity rules, along with suggested property equivalences. Such rules cause the current instances of the local database to bear additional information.

3. In response to a request for related information, the SA suggests remote types on which to define approximate similarity rules, along with suggested property equivalences. Such rules add related foreign types and their instances to the local database.

4. When the SA detects structural conflicts between a local type and a related remote type, it suggests to define descriptivity rules between them.

5. When the SA detects the COLLECTION-OF relationship between a local type and a related remote type, it suggests to define constituency rules between them.

5 Extending the RE sharing mechanism: Some proposals

In this section, I describe the extensions to the RE sharing mechanism that are necessary to implement the ideas above. I discuss the implementation of each of the object relationships distinguished above, and the sharing of complex objects.

5.1 Strict similarity

Strict similarity is supported by instance sharing. Remote instances \( \delta' \) that satisfy the condition \( \psi(\delta') \) under which they are defined to be strictly sim-
ilar to the local type $T$, are imported into that type. For example, all

Conference instances whose name contains “ACM” are to be imported into

ACM. Equivalent functions of $O$ and $T$ are dealt with through the standard

RPC calls of the surrogate mechanism. We do not assume that equivalent

local and remote functions have the same name; the RPC mechanism can

hide homonyms and synonyms. Instead of importing a remote function as

yielding values in the remote domain of the function, however, we can adapt

it to locally adhered domains by applying a conversion function $cf'$ to its

result, thus overcoming value incompatibilities.

Note that imported remote objects may support additional functions

that do not occur in type $T$. We have two options for dealing with such functions:

1. These functions may be collected in a new subtype $S$ of $T$ that is

   added to the local hierarchy, and contains the surrogates occurring in

   $T$ (do not confuse $S$ with the surrogate type for the imported objects,

   which is used by the RPC-overriding mechanism, and is invisible to

   the local user);

2. Alternatively, these functions may be added to the local type itself

   through behaviour sharing. This is especially attractive for computed

   functions. For stored functions, the local user is then responsible for

   providing values for these functions on the native objects.

5.2 Object identity

In the current version of RE, remote instances that are imported into the

local database are implicitly assumed to represent a real-world object that

is not represented in the local database yet. In my terminology: the object

is assumed to be strictly similar to the local class, but not identical to any

of the local instances.

However, in terms of our example, some of the publications contained

in researcher A’s database may already exist in researcher B’s database. Using

behaviour sharing, and by introducing some simple extensions to the

instance sharing mechanism, we can allow for the importation of identical

objects as well:

- A surrogate type is introduced for the identical remote objects. The

  type is completely identical to the type holding the surrogates for

  the strictly similar imported instances, except for the way duplicately

  named (stored) functions are overridden, as explained below.
Figure 2: Identity and strict similarity

- The local instance to which the imported instance is identical, is added as an instance of the new surrogate type. Note that we need not create a new surrogate here; instead it must be possible to dynamically assign an additional type to an instance.

- Overriding of local functions is now as follows. Instead of just calling the remote function through an RPC, as is done for strictly similar objects, the function is evaluated as \( df(cf'(RPC f), cf(lo e f unc)) \). Thus, the local and remote results of the function are resolved through the decision function as specified in the property equivalence assertion.

- Possible additional remote functions are dealt with as sketched under strict similarity.

Example: Identity vs strict similarity  Again, consider the example of Figure 1. Suppose we have the following object relationships: \( Eq(A6, B6) \) and \( Sim(A7, ACM) \). (Determining such relationships involves evaluating the conditions specified in the ocr’s against the state of these objects. For example, \( A7.Name() \) might be “ACM Tr. on SE”.)
Figure 2 illustrates how the objects $A_6$ and $A_7$ are imported into researcher B's database. Note that the local stored functions Abstract(), AppIn(), and Sig() are initialised to "" for $S_1$, which is the local surrogate for $A_7$.

The example illustrates both of the approaches to incorporating additional remote functions mentioned above. While the (computed) MakeRef function is shared, and added to the local type Publ, the additional remote functions TextBody() and View() are catered for using a new subtype of ACM. These functions are available for surrogate objects only.  

Object identity and the behaviour sharing taxonomy  

Note that this mechanism introduces a new type of behaviour sharing: importing a remote stored function to a local instance (the function TextBody() on $B_6$ in our example). Currently, this type of behaviour sharing is seen as 'somewhat meaningless'. We have just seen that this type of sharing does make sense if there exists an equivalent of the local instance in the remote database. Hence this type of sharing is the basis for the importation of remote instances that are identical to some local instance.

Note that I discuss sharing of computed functions, including the consequences of identity between local and remote instances, in a separate subsection.

5.3 Approximate similarity

In contrast with strict similarity and identity, the instances imported in the case of approximate similarity cannot be assigned to a type in the local hierarchy. Hence we resort to type sharing to support approximate similarity. Suppose we have an approximate similarity rule between the remote type $T'$ and the local type $T$. This can be supported by the sharing mechanism as follows:

1. Create a new local virtual supertype $S$ of $T$, containing the equivalent properties of $T$ and $T'$. $S$ does not have any direct instances.

2. Import the remote type $T'$ as a subtype of $S$.

3. Import all instances of $T'$ that satisfy $\xi(o')$, the condition under which $T'$-objects are approximately similar to $T$.

---

1 The difference between the computed functions View() and MakeRef is discussed in the section on behaviour sharing.
Approximate similarity does not occur in our example; it is straightforward to implement.

5.4 Structural differences (descriptivity)

A structural difference, or descriptivity relationship, between components occurs when there is an equivalence between a set of functions, each returning a simple type, occurring in one component, and a function returning a user-defined type in another component. In the context of the RE sharing mechanism, we must distinguish two cases:

1. The local type $LT$ contains the set of simple-valued functions; the imported remote type $RT$ contains an equivalent function returning the user-defined type $RUT$.

   The solution here is to equip the surrogate type of $RT$ with simple functions $f$ as in the local database, evaluated by performing an RPC that follows the corresponding path $RUT.f$ in the remote database.

2. A local type $LT$ contains a function $f$ returning the user-defined type $LUT$; the imported remote type $RT$ contains an equivalent set of simple-valued functions.

   The solution here is to create two surrogate objects for each imported instance object $o \in RT$: one ($s1$) to be imported into $LT$, and one ($s2$) to be imported into $LUT$. Both obtain their values from executing an RPC on the same remote instance. Moreover, $f(s1) = f(s2)$.

Note that structural differences cause us to change the one-to-one relationship between surrogates and remote objects into a many-to-one relationship.

**Example** Our example illustrates the second of the cases distinguished above. The local type `Publisher` does not have an equivalent in the remote database; instead, the remote type `Publication` contains a function `Publisher():String`. The extended sharing mechanism creates surrogate `Publication` instances from the remote `Publisher()`-values. Note that according to the ocr specified, such objects need to be created only if there does not exist a `Publisher`-object with the same name already; publishers with the same name are assumed to be the same. In other words, virtual remote `VirtPublisher` objects can have identity and similarity relationships with local objects too.
In our example, we assume that $A_6.Publisher() = B_1.Name()$; hence a surrogate need not be created. Possibly, however, this introduces a conflict in the $PublishedBy()$ value between $A_6$ and its local counterpart $B_6$, which might assign a different local object as its publisher. Such a conflict is then solved using the decision function for $PublishedBy()$ as specified by the interoperation designer. The value of $A_7.Publisher()$ is assumed to give rise to a new surrogate Publisher object $S_2$ (which, of course, returns """ for Location()). Note that surrogates $S_1$ and $S_2$ stem from the same remote object $A_7$. Note furthermore that since the remote “virtual objects” do not provide overlapping additional information for Publishers, a distinction between strictly similar and identical R-Virt-Publisher-objects need not be made.

5.5 Sharing complex objects

Complex objects are made up of component objects with relationships among them; object relationships are implemented by functions whose return type is a non-primitive one. Complex objects are an essential part of
the object-oriented approach, but many “OO” interoperability frameworks neglect them. Although in the documentation of the Sharing Advisor this subject is addressed at the type level, at the level of the actual sharing mechanism there is no concept of complex object sharing. Here I discuss how complex object sharing could be realised in the context of RE.

5.5.1 Importing related instances

Consider the general case of a complex object relationship, where a remote instance $o1'$ of type $T1'$ has a stored function $f'$ with type $set-off(T2')$ and value $S = \{o2'_1, o2'_2, \ldots, o2'_n\}$. Sharing of such a complex object involves sharing of the objects $o1', o2'_1, o2'_2, \ldots, o2'_n$ by creating local surrogates $so1, so2_1, so2_2, \ldots, so2_n$, respectively. Also, the function $f'$ must be shared.

How this is done depends on two factors:

- Whether or not relationship merging must be performed. Relationship merging occurs if there exists a local object $o$ which is identical to $o1'$, with a relationship implementing function $f$ which is equivalent to $f'$. $f$ and $f'$ must then be merged for $o$;

- Whether the relationship is shared in a materialised or a virtual way.

Virtual vs materialised relationship sharing The latter two modes of relationship sharing are now explained in the situation where relationship merging does not occur.

- Materialised relationship sharing refers to the situation where the function $f$ on $so1$ is defined as $\{so2_1, so2_2, \ldots, so2_n\}$. That is, the relationship is defined as is at the moment of importation from the remote source. Possible updates to the value of $f$ are now under control of the importing database; changes to the value of $f'$ are not reflected in the value of $f$.

- Virtual relationship sharing occurs when $f$ is defined as $Surr(RPC(f'))$, where $RPC(f')$ retrieves the value of $f'$ from the remote database, and $Surr(\{o'_1, o'_2, \ldots, o'_n\})$ returns the set $\{so1, so2, \ldots, so_n\}$ of local surrogates for the remote objects. The implementation of $Surr(S)$ requires a query of all surrogate objects whose $R-OID$ value occurs in $S$, and the creation of new surrogates for the remote objects for which a local surrogate does not exist yet.
Thus with virtual relationship sharing $f$ is locally implemented as a computed function, while with materialised relationship sharing $f$ is a locally stored function.

**Relationship merging** An interesting situation arises when in addition to the remote object $o_1'$ having a relationship with objects $\{o_2', o_2', \ldots, o_2''\}$ implemented in the function $f'$, there exists a local object $o_1$ with a function $f = \{o_2, o_2', \ldots, o_2''\}$, where $o_1$ is identical to $o_1'$, and $f$ is equivalent to $f'$. In order for this relationship to be shared, the decision function $df$ associated with the property equivalence assertion for $f$ and $f'$ must be taken into consideration. Two options seem reasonable:

- $df = \text{union}$

  Assuming virtual relationship sharing, this redefines $o_1.f$ as $o_1.f_{old} \cup \text{Surr}(\text{RPC}(o_1', f'))$. Note that the union operands need not be disjunctive; some objects in $o_1.f_{old}$ may be identical to those in $o_1'.f'$. In other words, objects are considered to be related if they are related in any of the databases.

- $df = \text{intersect}$

  Assuming virtual relationship sharing, this redefines $o_1.f$ as $o_1.f_{old} \cap \text{Surr}(\text{RPC}(o_1', f'))$. Here the set of related objects are exactly those objects of $o_1.f_{old}$ and $o_1'.f'$ that are identical. In other words, objects are considered to be related if they are related in both databases.

**Example** Again consider our example of Figure 1. Recall that we assumed the relationship $\text{Eq}(A6, B6)$. Furthermore, assume $A6.\text{Authors}() = \{A1, A2\}$, $B6.\text{Authors}() = \{B2, B3\}$, and the additional relationship $\text{Eq}(A2, B2)$.

The importation of $A6.\text{authors}$ into B’s database involves the creation of a surrogate $S3$ for $A1$. No surrogate needs to be created for $A2$, since the identical object $B2$ already exists in the local database. The result of applying $B6.\text{Authors}()$ immediately after importation is $\{B2, B3, S3\}$ if the decision function is $\text{union}$, and $\{B2\}$ if the decision function is $\text{intersect}$. See Figure 4.

### 5.5.2 Constituency

The final object relationship to be discussed, constituency, is related to the issue of complex object sharing in the sense that constituency refers to a
complex-object relationship between objects in different components. That is, if there exists an object comparison rule $Aggr(O, O') \leftarrow \phi(O', O)$ between a local object $O : T$ and a remote object $O' : T'$, $O'$ is to be imported as a part of a complex object rooted at $O$. This is implemented by importing $T'$ and its instances into the local hierarchy, and extending the type $T$ with a computed function $f : T'$, with body $\text{return(Surr}(|\phi(O', O)})$).

In our running example, the object relationships $Aggr(A4, B7)$ and $Aggr(A5, B7)$ (which might be determined based on a correspondence between their respective Booktitle() and Title() values, as indicated by the constituency rules) would lead to the definition of a stored function Chapters() on Book, where $B7.Chapters()= \{S4, S5\}$, the local surrogates for $A4$ and $A5$.

5.6 The behaviour sharing mechanism

Although this seems to be suggested by the current RE papers, not every behavioural object can be shared among components. One cannot apply any local computed function to a remote object (or vice versa), because such a function typically requires access to a given set of attributes, which may not be supplied for the remote (local) object it is applied to. To apply a function $f$ to objects of (original) type $T$, the set $F_f$ of functions called by $f$ should
be a subset of the functions defined on $T$.

**Example** Referring to our example of Figure 1, the function View() supplied by researcher A on his Publication objects cannot be applied to researcher B's Pub1 objects that have no equivalent Publication object in A's database. The View() function calls the stored function TextBody() on the objects it is applied to, and this function is not supplied by B's database. Since the importation of identical objects from A to B can be seen as supplying TextBody() values for B's objects, View() can be executed against those Pub1 objects for which an identical Publication object exists only. This is reflected in the type hierarchy of Figure 2 in that View() is defined on the subtype FullACM only.

A's MakeRef(Style) function, on the other hand, can be applied to B's Pub1-objects, since these provide all stored functions necessary to compute MakeStyle. Hence in Figure 2's type hierarchy MakeRef(Style) is imported as a function on Pub1.

**The USES clause** Hence the specification of the body of the shared function in addition to just its signature, as in TM-like formalisms, or at least a specification of the functions that are called by a computed function, is called for. In the example, I have added the USES clause to the declaration of computed functions.

The necessity for shared computed functions to access the state of the objects they are computed on is acknowledged by the creation of the *callback* mechanism, but we need to know on beforehand which functions a ‘callback’ will be made for, and it must be assured that these are indeed available.

## 6 Conclusion

In this document, I discussed possible extensions to the RE framework, which stem from my own previous research. It appears that RE is a suitable environment for instance-based database interoperation, and that its sharing mechanism can easily be extended to fully support instance-based database interoperation specifications consisting of object comparison rules and property equivalences. Object comparison rules thus act as an interface between the Sharing Advisor and the sharing mechanism.
References


