

A Generative Communication Service for Database Interoperability

Wilhelm Hasselbring

*INFOLAB, Department of Information Management and Computer Science
Tilburg University, 5000 LE Tilburg, The Netherlands
e-mail: hasselbring@acm.org*

Mark Roantree

*School of Computer Applications
Dublin City University, Dublin 9, Ireland
e-mail: mark.roantree@compapp.dcu.ie*

Abstract

Parallel and distributed programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to manage the coexistence and coordination of multiple concurrent activities. The model of ‘Generative Communication’ in Linda — a paradigm that has been developed for parallel computing — emphasizes the decoupling of cooperating parallel processes; thus, relieving the programmer from the burden of having to consider all process inter-relations explicitly.

In many application areas, data is distributed over a multitude of heterogeneous, autonomous information systems. These systems are often isolated and an exchange of data among them is not easy. On the other hand, support for dynamic exchange of data is required to improve the business processes. Cooperative information systems enable such autonomous systems to interoperate. They are complex systems of systems which require a well designed and flexible software architecture.

The Linda model had a great influence on research in parallel programming languages. Stimulated by this success, a Generative Communication Service, which offers a very flexible associative addressing mechanism based on metadata matching, has been developed for supporting interoperability of cooperative information systems. Some design patterns guided the construction of the resulting communication service that has been implemented on top of CORBA for an ODMG canonical data model.

Keywords: Interoperability, Multidatabase Systems, Generative Communication, Communication Service, Design Patterns, Linda, CORBA, ODMG.

1. Introduction

Cooperative information systems are complex *systems of systems* which require a well designed and flexible software architecture. This paper presents a Generative Communication Service that has been developed, based on the experience with object-oriented communication frameworks and the Linda generative communication model. The experience using design patterns in the development of an object-oriented communication framework guided us in structuring the communication service. The resulting Generative Communication Service offers a very flexible associative addressing mechanism based on *metadata matching*, i.e. it aims to support interoperability of cooperative database and information systems. We base our work on previous research in parallel programming [7, 11], and reuse some of our previous experiences with object-oriented communication frameworks [10] and the use of metadata in multidatabase systems [17] to design the Generative Communication Service.

Our research is based in a healthcare multidatabase environment. We assume a multidatabase or federated database system to refer to a loose coupling of participating heterogeneous database systems. We will use the terminology in [16] and [18] when referring to the multidatabase architecture and components. We have the following problem to consider: two autonomous databases have a requirement to exchange information in a multidatabase environment. They elect to pass information objects using a communication service that should not need to know anything of the makeup of the participating databases (their schemas, etc.). However, if different kinds of information (represented as objects) should be transferable, then some mechanism for handling them must be available at the communication service.

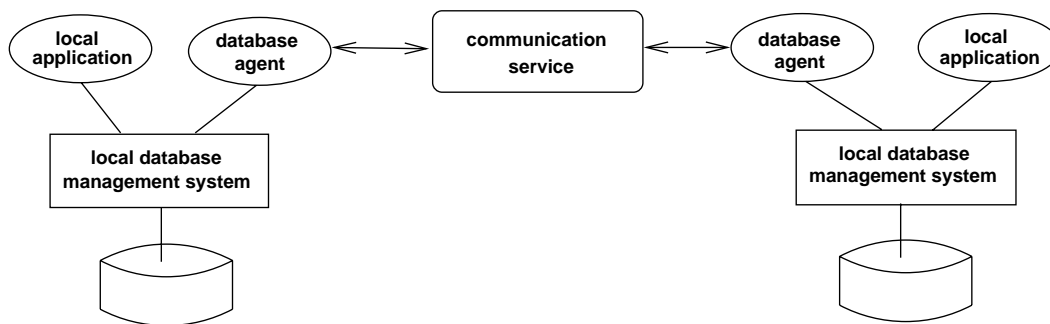


Figure 1. The general system architecture with database agents as mediators between local database systems. The database agents exchange information through a communication service.

Let us consider the general system architecture. To achieve a division of labor between system components, database *agents* should be connected to the local database systems to serve as mediators. The database agents transform the data between the local data models and the canonical data model (we use ODMG-93 [3]) in the sense of a federated schema architecture [18]. The communication service that manages the information exchange, sees these database agents as *active* database systems that exchange information on their own initiative. An active database system is an extended database system which has the capability to monitor predefined situations (situations of interest) and to react with defined actions [20]. Figure 1 displays the general system architecture illustrating the division of labor between the communication service and database agents. From the local database management system's point of view, the agents are local applications.

The problem is how to transfer information such that the individual database systems do not need to know the other systems or how many other systems are connected. The systems should only say what they offer (i.e., are willing to send) and what they need (i.e., want to receive). To achieve this flexibility, we developed a communication service under the guidance of some design patterns and combined this software architecture with the decoupled communication model of Linda, which is called *generative communication* [7].

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [6]. Design patterns can be specific enough to name particular objects, their responsibilities, and interaction. A well-known pattern of this kind is the Observer pattern from [6, pages 293ff]. It supports keeping cooperating components consistent, with the help of a change propagation mechanism. Another design pattern is called *Prototype* [6, pages 117ff]. The basic idea of this design pattern is that the different kinds of objects

which need to be constructed, are represented through 'prototypical' instances that are able to 'clone' themselves by copying the corresponding prototype.

This paper presents a Generative Communication (GC) service which operates as a *prototype-factory service* in a CORBA environment. It has been designed for accomplishing the transfer of information among interoperable database and information systems, such that:

- The GC service does not need to know the structure and different types of information to be transferred in advance, nor does it need to define these objects in its type hierarchy. It only manages the descriptions (metadata) of the information to be exchanged in a prototype factory.
- The individual information systems do not need to know each other. It is required that they agree on the structure of information (metadata) they intend to exchange.

Section 2 discusses some previous and related work on object-oriented communication frameworks, generative communication, and metadata modeling in multidatabase systems to serve as a motivation for the generative communication service that is presented in Section 3. Section 4 summarizes the paper and indicates areas for future work.

2. Previous and Related Work

Subsection 2.1 first discusses some previous work on an object-oriented C++ communication framework. A note on the terminology: The communication *framework* discussed in Section 2.1 is a C++ class hierarchy together with models of interactions which can be turned into complete applications by creating specializations which concentrate on a more concrete task. Such an architecture is usually called

‘object-oriented framework’ [4]. The communication *service* presented in Section 3 offers some CORBA objects to the programmer. Such CORBA objects provide *services* to other CORBA objects [14].

Subsection 2.2 then explains Linda’s model of *generative communication* that has been developed for parallel programming to serve as a motivation for the generative communication service. Another note on terminology: A *concurrent* program specifies two or more processes that cooperate in performing a task [1]. Each process is a sequential program that executes a sequence of statements. Processes cooperate by communication and synchronization. In a *parallel* program, these concurrent processes are executed in parallel on multiple processors. A *distributed* program is a concurrent program in which processes on different computers communicate through a network. CORBA is usually applied in distributed programming. The resulting combination of generative communication and CORBA services is a *Generative Communication Service*.

Subsection 2.3 discusses some concerns of metadata modeling in multidatabase systems.

2.1. Communication Frameworks

A C++ communication framework that has been developed as part of a larger project in which heterogeneous information systems needed to interoperate within hospitals is presented in [10]. An important goal for the system design was to decouple the subsystem components simply such that individual subgroups within the project team could work independently, while agreeing on small interface specifications. The federated system architecture in this project has been designed according to the specific requirements of integrating replicated information among heterogeneous systems within hospitals [9].

Object-oriented frameworks can be regarded as incomplete software architectures which can be turned into complete applications through various kinds of specialization [4]. Design patterns guide the construction and documentation of frameworks, but they may also be *discovered* in existing object-oriented frameworks, e.g., in frameworks for graphical user interfaces, communication middleware, databases, etc.

A crucial design decision was made regarding the architecture of the communication framework. The resulting C++ communication framework encapsulates the CORBA services to exchange information. A basic problem was how to transfer information using the communication framework, in a way in which it does not need to know the internal structure of data objects and in which the cooperative information systems do not need to know the communication platform (CORBA, in this case).

The first step for designing the communication frame-

work was to base it on the design pattern *Abstract Factory* [6, pages 87ff]. This pattern permits users of a factory to obtain an abstract interface for creating families of related objects without specifying their concrete classes. With the *Abstract Factory* pattern the communication framework does not need to know the concrete classes it is required to transfer. Every product (information object) in the product family is encoded within the model and the program code. In case of a requirement for additional types of information, it is necessary to modify the communication framework as a client of the factory; thus, yielding a somewhat inflexible design.

This situation led us to search for a solution in which the communication framework becomes decoupled from changes with respect to the structure *and* the number of different types of information objects. The next step was to employ the design pattern *Prototype Factory* [6, pages 117ff]. This allows different classes of information objects and their handlers being represented through ‘prototypical’ instances that are able to ‘clone’ themselves [10].

It transpired that the developed communication framework is an object-oriented framework with *inversion of control* [4]: the framework calls the application which uses the framework. Event handler objects of the application are invoked via the framework’s reactive dispatching mechanism. The handlers that represent the application logic for processing received information are called by the communication framework. This is different to the reuse in procedural languages such as C, where the application calls functions/procedures which are provided by a library. We can only present an coarse overview of this architecture in this paper. For a more detailed description refer to [9].

With this architecture, the information can be transferred through the communication framework in a way that

- the framework does not need to know the structure and different types of information to be transferred and
- the individual information systems do not need to know the employed communication platform.

This way, it was feasible to decouple the system components in a flexible way such that the individual subgroups within the project team were able to work independently, while agreeing on small interface specifications.

However, with this approach, both sender and receiver of information must agree on the same concrete C++ classes of information objects and they must know each other to exchange information objects via send and receive operations (message passing). To achieve a flexible decoupling, we developed a new communication service based on Linda’s *generative* mechanisms (see Section 2.2) combined with the positive experience with the *prototype-factory* pattern that was employed in the previous project’s C++ communication framework. In the follow-up project, which is discussed in

Section 3, we decided not to hide CORBA, but to deliver the new communication service as CORBA service objects. This way, it is possible to combine the new communication service with other CORBA services such as events and security. An additional advantage with the integration into CORBA is the possibility to connect components that are implemented in different programming languages.

2.2. Generative Communication in Linda

There has been particular attention to parallel computing within the computer science community over the last twenty years. Many programming models and languages have been developed for parallel programming. However, for many application areas, the often used parallel-programming model of *message passing* is too low-level and inflexible [19]. Therefore, several alternative models have been designed for parallel programming, which provide higher-level abstractions. These languages emphasize some element of shared data.

The Linda programming language [7] is an example of significant advance in this area. The shared data pool employed in Linda is called *tuple space*, and is a collection of tuples. All Linda communication is a three-party operation: sender interacts with tuple space, tuple space interacts with receiver. Conversely, traditional models such as point-to-point message passing provide two-party operations. Process communication in Linda is called *generative communication*, because tuples are added to, removed from, and read from tuple space [7].

Reading access to tuples in tuple space is associative and not based on physical addresses. This method is somewhat similar to the value-based selection of entries from a relational database. Tuples in tuple space are selected by *matching*, whereby a tuple and a template are defined to match, iff they have the same structure (corresponding number and type of components) and the values of their actuals are equal to the values of the corresponding tuple fields.

Linda defines some operators, which may be added to a sequential computation language. These operators enable sequential processes, specified in the underlying computation language, to access the tuple space. The `out` adds tuples to the tuple space. The `in` operation attempts to withdraw a tuple from tuple space. Tuple space is searched for a matching tuple against the template supplied as the operation's argument. If and when a tuple is found, it is withdrawn from tuple space, and the values of its actual fields are bound to any corresponding formals in the template. Tuples are withdrawn *atomically*. If no matching tuple exists in tuple space, the process executing the `in` suspends until a matching tuple becomes available. If many tuples satisfy the match criteria, one is chosen arbitrarily. The `rd` operation is the same as `in`, with actuals assigned to for-

mal as before, *except* that the matched tuple remains in tuple space. Additionally, many Linda dialects provide non-blocking extraction operations, multiple tuple spaces and support for process creation. C was the first computation language in which Linda was integrated. In addition, high-level language integrations exist which support early phases in software development, such as prototyping for early design evaluation [11].

Linda's shared, associative object memory supports a highly *decoupled* programming style in which processes remain mutually anonymous. This enables the programmer to focus on one process at a time; thus, making parallel programming conceptually the same order of problem-solving complexity as conventional, sequential programming. With generative communication each access to shared data is asynchronous: sender and receiver of a tuple do not have to exist at the same time, and do not have to do things synchronously.

This paper suggests learning from the experience with parallel computing and applying some of the ideas of the flexible Linda *parallel* programming model — in particular the decoupled communication enabled by matching — to *distributed* programming as it is required for database interoperability.

2.3. Metadata Modeling in Multidatabase Systems

Metadata plays an important role in information systems interoperability [13]. We have attempted to combine the ideas of object-oriented frameworks and generative communication with previous research undertaken on metadata modeling in multidatabase systems [17]. In this research one of the functions of the canonical model was to represent the *virtual* schemas (export and federated schemas [18]) as metadata components. Since CORBA objects must be compiled before they are used, it was necessary to build flexible objects which could accommodate change without the need for recompilation. This was accomplished through the usage of generic classes. This use of generic classes is now extended in this paper where prototypes are used to describe any ODMG [3] object types, and clones can be constructed from these prototypes to carry data values.

3. A Generative Communication Service

In this section we discuss the design for a Generative Communication (GC) Service which is based on the concept of a *Prototype Factory* model (described in Section 2.1) and generative communication (described in Section 2.2). Let us consider the problem discussed in the introduction where database *A* wishes to send some information in the form of objects to database *B*. These databases either have similar

schemas, both type hierarchies contain the objects they wish to share, or they have agreed a common format in advance for transferable objects. We propose a system whereby an intermediary service, the GC service, is used to define prototypes of these objects, and then to create a clone for every information object it is required to store and forward. Thus, if *A* wishes to send a set of objects to *B*, it does so by checking to see if a prototype has been created previously with the GC service. If a prototype exists, object clones can be defined, otherwise object clones can be created for every object *A* wishes to send. The design model for the GC service contains two main object types: the **Prototype** object type and the **Clone** object type which are used to describe objects discovered at runtime, and carry actual data values respectively.

The GC service is based on the *Prototype Factory* pattern where object prototypes are defined for unknown object types. A registration process provides the service with a description of the class of objects it is required to manage. When it is necessary to create and store an instance of this class, a clone of the registered class is constructed which holds actual data values. This work is similar to the prototype factory discussed in Section 2 in that we can handle objects of unknown type; however, we use a metamodel to describe these objects that allows us to compare object definitions. In other words, it is necessary for the GC service to be able to understand the internal structure of these objects. This provides us with a useful matching technique for object prototypes which we will discuss later in this section.

3.1. Metamodel Description

A metamodel can be described as being an abstraction of metadata, which in turn describes the model. We assume that our multidatabase system uses an ODMG object model as a common model for participating information systems. Thus, the requirement for our metamodel is the manipulation of ODMG metadata. More specifically, we have a requirement to define ODMG types and describe ODMG constructs such as classification, inheritance, and relationships. As we shall explain later, we do not have a requirement to model behavior. The metamodel description provided here is not detailed as we are concerned with the metadata of an existing, well-known object model [3].

Before describing the GC service it is necessary to understand the **Prototype** class used to describe prototype objects. As the GC service must carry objects of different and unknown types it must first create a prototype object. This is achieved by creating a new instance of the **Prototype** class and then using the registration operation (through the **Register** method) to describe the prototype object. In essence, the **Prototype** class is a ‘template’ class, which describes new types discovered at runtime, and contains some

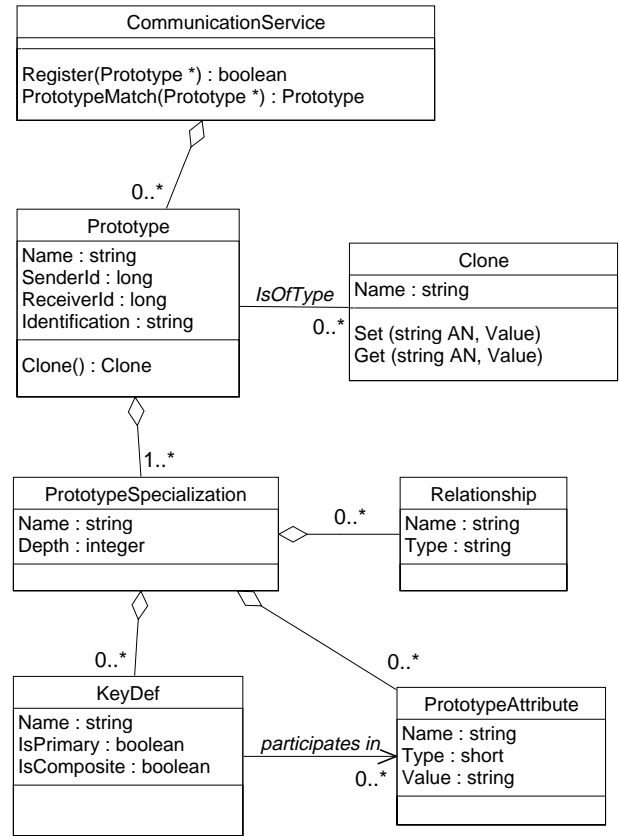


Figure 2. The metamodel of the Generative Communication Service modeled in the UML notation for class diagrams [5]. Hollow diamonds indicate part-of relations (aggregation).

data elements (attributes used by naming and security services) and some metadata elements. This closely relates to the Linda model of formal and actual parameters described in Section 2.2 where formal parameters are metadata elements and actual parameters are data elements. Our metamodel (illustrated in Figure 2) is implemented through the **Prototype** and **Clone** classes with their contained metadata elements. The metadata element comprises a series of **PrototypeSpecialization** objects of which each **Prototype** object must have at least one. They are used to describe the inheritance hierarchy of the object type to be registered. The set of subclasses are represented as an ordered list of objects containing class names and a **Depth** attribute which is used to determine the order of subclasses. It is assumed that all objects inherit from the root **Object** class, which is an instance of the class **PrototypeSpecialization**, and it is thus, only necessary to describe all specializations of the **Object**

class when registering the new type with the GC service. All `Prototype` objects are *contained* within the `CommunicationService` object, which is the CORBA service object.

Each `PrototypeSpecialization` has n attributes ($n \geq 0$), m relationships ($m \geq 0$) and p candidate keys ($p \geq 0$). Candidate key elements must map directly to one or more attributes defined for the new object type. Since both suppliers and consumers contain transferable types in their local type hierarchies, it is not necessary to transfer behavior and thus, it is not part of our metamodel. It is also assumed that consumers are aware of which behavioral parts are no longer valid. For example, if the supplier is exporting only a projection of the overall object, then it is possible that not all methods will function on the object's projection. The supplier will be aware of the projection and can determine which behavioral attributes are no longer valid. The `PrototypeSpecialization` and `Relationship` classes are used to create the structure for each of the classes described in the inheritance hierarchy. *Names* and *types* are required for both `PrototypeAttribute` and `Relationship` objects. The `Type` attribute in the `Relationship` class is used to determine whether the relationship is a 1 – 1 or 1 – n (set) relationship. Finally, the `KeyDef` class is used to describe keys for each of the information classes. Note that in the implementation, class names may change as separate classes are used for prototypes and clones (see below).

Once a class has been registered with the GC service the `Clone` method is used to create an instance of these objects. We will shortly discuss how actual data values are transferred between supplier and consumer database agents and the GC service. The `CommunicationService` class also has a `PrototypeMatch` method which is used for suppliers to match their 'query' against information objects held by the GC service. This process is explained in the following section.

3.2. Implementation Details

Our first version of the GC service was developed in C++ using Orbix [2], an implementation of CORBA. CORBA is the 'Common Object Request Broker Architecture' of the Object Management Group, used to standardize interoperability among heterogeneous hardware and software systems [14]. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA defines an Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA also defines interoperability by specifying how ORBs from different vendors can interoperate.

Our architecture assumes a federation of databases

```
interface Clone
(
    extent Clones
    key name)
{
    attribute string name;
    relationship <Prototype> IsOfType
        inverse Prototype::Memberset;
    relationship set <CloneSpecialization>
        Subclass inverse CloneSpecialization::
            ClassType;

    boolean Set(in String AttributeName,
                in String Value);
    boolean Get(in String AttributeName,
                out String AttributeValue);
};

interface Prototype;
(
    extent Prototypes
    key name)
{
    attribute string name;
    attribute long sender;
    attribute long receiver;
    attribute string identification;
    relationship set <Clone> MemberSet
        inverse Clone::IsOfType;
    relationship set <PrototypeSpecialization>
        SubClass inverse
            PrototypeSpecialization:: ClassType;

    Clone clone(void);
};
```

Figure 3. Extract of the ODL interfaces for clones and prototypes.

which use an ODMG object model as their canonical data model. We assume that some ODMG compliant database agents (providing component schemas in the traditional federated database sense [18]) wish to exchange information (see Figure 1). In this illustration, the data model translation of local database schema to the canonical schema is encapsulated inside the agent process at each local database. The agent presents the component schema to the federation. CORBA objects are used to provide an interface to each of the database agents and to provide distribution for the architecture.

When a consumer connects to the GC service it will pass a query in the form of 'do you have clones with this structure?'. The assumption is that a consumer knows (from

previous agreements with suppliers) that objects of certain types will be transferred. The request to the GC is not based on data values but on object types. At this stage, it is only necessary to retrieve objects which are of a specific type. This is achieved through the prototype matching operation `PrototypeMatch`. The consumer must define an object prototype which may be transient or persistent. Persistent definitions are used when the consumer makes regular requests for the same type of data, and are stored in the GC service database in the same manner as supplier prototypes. Once the consumer has defined a prototype, the `PrototypeMatch` method is used to detect any prototypes (and subsequently their associated clones) which match this type. This is necessary as it is impractical to arrange a naming scheme for prototypes in advance, when using a concurrent service with possibly hundreds of users. It also means that supplier and consumer have no need to communicate once an initial arrangement has taken place on the types of data to be transferred. (Note that if the results are presented in some form of a view, a subsequent OQL-type query can refine this data set to those objects which match a certain criteria. This makes it possible to download only a subset of the objects.)

In Linda, the matching of more than one prototype results in an arbitrary selection of a single prototype. However, we use the `identification` attribute in the `Prototype` class to inform suppliers of the origin of the prototypes. If the `identification` can be decrypted to provide a meaningful term, then the consumer knows it has the correct prototype. To automate this, it is necessary for all identification labels to translate to the same 'known' term to all suppliers. We are currently investigating ways of improving this although it is not possible to use data values in the clone objects as security is vital in healthcare systems, and access to data is restricted to consumers who have appropriate decryption keys.

The following ODL type definitions (ODMG-93 [3]) help to illustrate the differences between prototype and clone objects in the implementation (Figures 3–5). Although prototypes and clones are very similar in structure (it could be argued that a prototype is a specialization of a clone), it was decided to treat them as separate object types due to the fact that there was not enough reusable elements between them, as displayed in Figure 3. The `Prototype` and `Clone` objects are the primary objects in the GC service database; the remaining objects cannot exist without a reference to a prototype or clone object.

As `Clone` metadata objects are identical in structure, it is necessary to associate each collection of `Clone` objects with the corresponding `Prototype` object. This is achieved in both directions: the *IsOfType* relationship in the `Clone` class maps each clone object to a prototype object; the `Memberset` relationship in the `Prototype`

```
interface CloneSpecialization
(
    extent CloneSubclasses
    key name)
{
    relationship <Clone> ClassType
        inverse Clone::Subclass;
    relationship set <CloneAttribute>
        AttributeSet inverse
            CloneAttribute::Specialization
}

interface PrototypeSpecialization
(
    extent PrototypeSubclasses
    key name)
{
    attribute string name;
    attribute short depth;
    relationship <Prototype> ClassType
        inverse Prototype::Subclass;
    relationship set <PrototypeAttribute>
        AttributeSet inverse
            PrototypeAttribute::Specialization
}
```

Figure 4. ODL interfaces for specializations.

class maps each prototype object to a set of clone objects.

The inheritance hierarchy is constructed using classes to represent specialization. Once again, it was decided to use separate `Prototype` and `Clone` classes to model the hierarchy description, and the hierarchy of data values respectively. `Clone` and `prototype` objects contain a relationship to their inheritance hierarchy through the `SubClass` attribute. The link between a set of subclasses and the prototype or clone objects is achieved through the `ClassType` attribute (see Figures 3 and 4).

As each specialization (or subclass) will contain a set of attributes, the `AttributeSet` attribute in both of the above classes provides a relationship to a set of objects, which describe the attributes in the case of `PrototypeSpecialization` classes, and contain data values in the case of `CloneSpecialization` classes. The `CloneAttribute` class must be able to maintain an array of values to handle possibilities such as arrays of values, a set of relationships (oids) and ODMG collections such as tuples, bags and sets.

The final ODL sample in Figure 5 illustrates the `CloneAttribute` and `PrototypeAttribute` classes, together with the `KeyDef` class which is used by prototypes to model database keys. In essence, `CloneAttribute` objects contain data values and a set of methods (not shown) to convert between the stored string value

```

interface CloneAttribute
( extent CloneAttributes)
{
  relationship <CloneSpecialization>
    Specialization inverse Specialization::
    AttributeSet;
  attribute array <string> AttributeValue;
}

interface PrototypeAttribute
( extent PrototypeAttributes)
{
  attribute string name;
  relationship <PrototypeSpecialization>
    Specialization inverse Specialization::
    AttributeSet;
  attribute short AttributeType;
  attribute string AttributeValue;
  relationship <KeyDef> KeyValue
    inverse KeyDef::Keyset;
}

interface KeyDef
( extent keys
  key name)
{
  attribute string name;
  relationship set <Attributes>
    Keyset inverse Attributes::KeyDef;
}

```

Figure 5. ODL interfaces for attributes.

and the actual data type. The `AttributeType` attribute in the `PrototypeAttribute` class contains an enumerated type denoting system data types, a relationship, a relationship set, or an ODMG collection.

3.3. A Sample Transfer Operation

In this section we will demonstrate how the GC service operates in an environment where two healthcare software systems share information. In the sample database view in Figure 6 it is intended to export details of all HIV patients who have `bloodtype 'O'` using the partial type hierarchy in the illustration. The first step is to register both object types with the GC service. A validation layer verifies that relationship attributes are valid. For example, clones cannot be created for the `Person` prototype unless the `Address` prototype is also registered.

The registration process first requests a name, `senderid`, `receiverid` and number of subclass levels in the object hierarchy. The `senderid` and `receiverid` are optional in our current implementation as no security layer exists. It is then necessary to register

the `Person`, `Patient` and `HIV_Patient` subclasses. When attributes have been defined for a sub-class, candidate keys can be defined based on the attribute set described for each class. As already stated, the `Address` prototype must also be registered before any clones can be constructed.

Once the classes have been registered, the `clone` operation is used to create an instance of one of these classes. The current implementation places the onus on suppliers to construct clones where required and populate them with data values. Each clone object has a `Set` method which is passed an attribute's name and value. It is assumed that suppliers create an export function which calls the `Set` method to write to the clones. In reality, this is part of the agent process. The code for each patient may look something like:

```

myclone.set("LastName", lname);
myclone.set("FirstName", fname);
myclone.set("dob", dob);
myclone.set("address", Address);

```

Note that in the above example, the onus is on the supplier to ensure the integrity of relationships (through the use of oids). It is necessary to create the `Address` clone first, obtain the appropriate oid, and pass this *Prototype Factory* oid to the `Patient` clone, rather than the original oid contained in the supplier's database.

A similar process takes place on the consumer's side where an import procedure must be constructed to query clone objects using `Get` to retrieve data values and populate objects in the consumer database. Our implementation assumes that consumers send a message to the GC service to inform it that data has been successfully transferred, which permits either the GC service or the original suppliers to destroy unwanted clone objects.

3.4. Synchronization

The CORBA event services are used in combination with our GC service to provide synchronous or asynchronous transfer of information objects using event channels [15]. Agents can either receive notification of events that concern them (*push* model) or can connect to the event channel to wait for their events (*pull* model). Figure 7 illustrates both mechanisms. The event service is implemented as a specialized CORBA object which means that it can be used by multiple suppliers and consumers simultaneously. In effect, this means that multiple suppliers can pass information to multiple consumers using the same event channel without any supplier or consumer having direct knowledge of each other. We use the OrbixTalk implementation of the CORBA event services [12], and the result is a flexible coordination architecture.

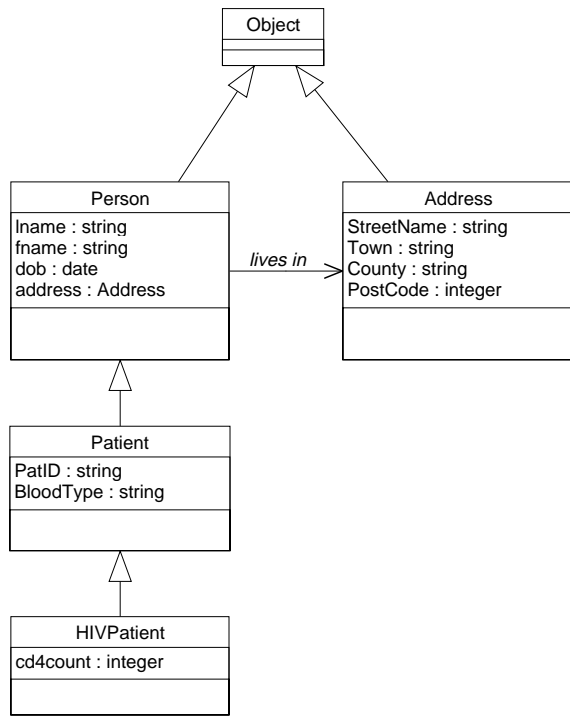


Figure 6. A sample healthcare object to be passed using the Generative Communication service.

3.5. Security

Our effort involved the implementation of a layer above the CORBA event services to permit the creation of prototype objects which describe *forthcoming information objects*, and the creation of clones to carry actual data values. A security layer is required to ensure that consumers only retrieve the information which is destined for them. The security layer is part of a future revision, where the GC service provides a table of public keys for each potential consumer. This has the effect that consumers can only decrypt messages which were meant for them, and were encoded using their public key. It also means that if suppliers can share encryption keys, it is possible to supply many users with a single broadcast. A more detailed discussion of security aspects is beyond the scope of the present paper.

4. Summary and Future Work

This paper starts with a discussion of previous and related work on object-oriented communication frameworks and generative communication in parallel programming, before

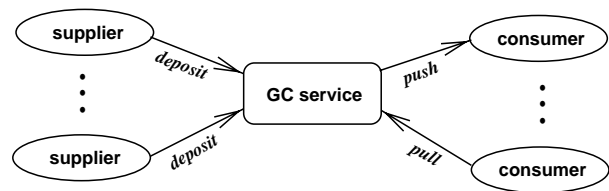


Figure 7. Push and pull communication through the GC service.

our new Generative Communications (GC) service is presented. To transfer information with this GC service, the following steps should be performed by the database agents:

- Senders and receivers agree on some metadata structure and the associated information contents. Pre-initialization of prototype objects that describe information objects to be transferred is done on registration of prototypes at the GC service.
- The prototype descriptions will later be used by consumers to match their information needs.
- Information transfer is accomplished as follows:
 - The sender sends an information object to the GC service which transfers data values to a clone object that represents the information object in the GC. The GC service acts as a buffer. The objects in the buffer are clones of the pre-initialized prototype objects.
 - Consumers can now receive the information object (which is a clone) from the GC service. The selection is based on formal parameters similar to the Linda model. The event service (OrbixTalk) wakes up the receiver on availability of the requested information.

Sender and receiver do not need to know each other and they do not need to exist at the same time: communication is asynchronous. This decoupling alleviates distributed programming. This fact is known from the experience with parallel programming, in particular with the Linda model for parallel programming.

The present paper also discusses how design patterns guide the construction and documentation of the GC service. With the presented architecture, the information can be transferred through the GC service in a way that:

- The GC service does not need to know the structure and different types of information to be transferred in advance. It only manages the descriptions (metadata) of the information to be exchanged in a prototype factory.

- The individual information systems do not need to know each other. It is sufficient to agree on the structure of information (metadata) they intend to exchange.

In this way, we achieve a feasible decoupling of system components in a flexible way.

For the discipline of software engineering, modifiability and extensibility (for maintenance) are important qualities to be adhered to in system's design [8]. One output is that the GC service can be re-used for other systems with similar communication requirements, particularly exchange of information among cooperative information systems.

One of the limitations of the current prototype is the difficulty in measuring the performance as the databases in our multidatabase prototype are small. In a real multidatabase system, some databases may be overloaded or simply unavailable. However, planned work includes a distributed query service which makes use of the GC service where nodes need to communicate to construct query subsets. This will provide an opportunity to evaluate the performance of the GC service as a central part of the multidatabase architecture.

Additional future work in this area is focused on the construction of a security layer. As we operate in a healthcare environment, security is crucial when transferring information in this manner. We are also improving the manner in how we handle a situation, where a consumer's query results in more than one prototype match. Finally, we are attempting to improve the metamodel to allow a more seamless transfer of data between the GC service clones and the supplier/consumer objects, particularly objects with more complex structures. At present, the onus is placed on suppliers and consumers to manage this transfer, with limited assistance by the service (in the form of `Get` and `Set` methods).

Acknowledgements

The first author is partly funded by the RENOIR network of excellence established within the Fourth Framework Programme for research and technology development in information technology (ESPRIT) of the European Union. The second author is funded by the Forbairt Strategic Research Grant Scheme.

Finally, the authors would like to acknowledge the useful comments made by the anonymous referees.

References

- [1] G. Andrews. *Concurrent Programming*. Benjamin / Cummings, Redwood City, CA, 1991.
- [2] S. Baker. *CORBA Distributed Objects, using Orbix*. Addison-Wesley, Harlow, England, 1997.
- [3] R. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufman, San Francisco, CA, 1996.
- [4] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997.
- [5] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, Reading, MA, 1997.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.
- [7] D. Gelernter. Generative communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, Jan. 1985.
- [8] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [9] W. Hasselbring. Federated integration of replicated information within hospitals. *International Journal on Digital Libraries*, 1(3):192–208, Nov. 1997.
- [10] W. Hasselbring. Design of a communication framework for interoperable information systems. In *Proc. Third World Conference on Integrated Design & Process Technology (IDPT'98)*, Berlin, July 1998.
- [11] W. Hasselbring. The ProSet-Linda approach to prototyping parallel systems. *The Journal of Systems and Software*, 1998. (in press).
- [12] IONA Technologies PLC., Dublin, Ireland. *White Paper – OrbixTalk*, 1997.
- [13] B. Krämer, M. Papazoglou, and H.-W. Schmidt, editors. *Information Systems Interoperability*. Wiley, 1998.
- [14] T. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley, New York, 1995.
- [15] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Object Survival Guide*. Wiley, New York, 1996.
- [16] E. Pitoura, O. Bukhres, and A. Elmagarmid. Object orientation in multidatabase systems. *ACM Comput. Surv.*, 27(2):141–195, June 1995.
- [17] M. Roantree, P. Hickey, A. Crilly, J. Cardiff, and J. Murphy. Metadata modelling for healthcare applications in a federated database system. In O. Spaniol, C. Linnhoff-Popien, and B. Meyer, editors, *Trends in Distributed Systems: CORBA and Beyond, International Workshop TreDS '96*, volume 1161 of *Lecture Notes in Computer Science*, pages 71–83, Aachen, Germany, Oct. 1996. Springer-Verlag.
- [18] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.
- [19] D. Talia. Parallel computation still not ready for the mainstream. *Commun. ACM*, 40(7):98–99, July 1997.
- [20] J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, 1996.