A Generative Communication Service for Database Interoperability

Wilhelm Hasselbring*
Department of Computer Science, University of Dortmund
D-44221 Dortmund, Germany, tel ++49 231 755-4712, fax -2061
e-mail hasselbring@acm.org

Mark Roantree
School of Computer Applications, Dublin City University
Dublin 9, Ireland, tel ++353 1 704-5636, fax -5442
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.

*New address from August 1998 onwards: Department of Information Management and Computer Science, Tilburg University, 5000 LE Tilburg, The Netherlands.

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 [12, 16], reuse some of our previous experiences with object-oriented communication frameworks [15] and the use of metadata in multidatabase systems [26] 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 [27] and [24] 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 wish 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.

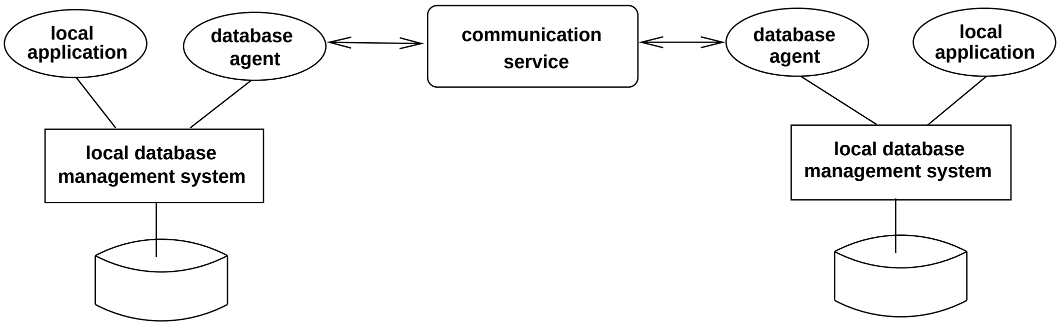

Figure 1. The general system architecture with database agents as mediators between local database systems. The database agents exchange information through a communication service.

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 [7]) in the sense of a federated schema architecture [27]. 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 [30]. 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* [12].

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [11]. 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 [11, pages 293ff]. It supports keeping cooperating components consistent, with the help of a change propagation mechanism. Another design pattern is called *Prototype* [11, 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 sufficient to agree on the structure of information (metadata) they intend to exchange.

Section 2 discusses some previous and related work on an object-oriented communication framework. A note on the terminology: The communication *framework* discussed in Section 2 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' [9, 25]. The communication *service* presented in this paper, offers some CORBA objects to the programmer. Such CORBA objects provide *services* to other CORBA objects [22].

Section 3 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 that is presented in Section 4. 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*. Section 5 summarizes the paper and indicates areas for future work.

2. Previous and Related Work

A C++ communication framework that has been developed as part of a larger project in which heterogeneous information systems needed to interoperate is presented in [15]. An important goal for the system design was to decouple the subsystem components in a simple way such that the individual subgroups within the project team were able to work independently while agreeing on small interface specifications.

Object-oriented frameworks can be regarded as incomplete software architectures which can be turned into complete applications through various kinds of specialization [9, 25]. Design patterns guide the construction and documentation of frameworks [19], but they may also be *discovered* in existing object-oriented frameworks, e.g., in frameworks for graphical user interfaces, communication middleware, databases, etc.

The larger project in which this communication framework has been developed aimed at integrating cooperative hospital information systems. The federated system architecture in this project has been designed according to the specific requirements of integrating replicated information among heterogeneous information systems within hospitals [14].

A crucial design decision was 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 by the communication framework, in a way in which it does not need to know the internal structure 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 framework was to base it on the design pattern *Abstract Factory* [11, pages 87ff]. The basic idea of this design pattern is that users of a 'factory' 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. However, the number of different products (information objects) in the product family is encoded within the model and the program code. In case of a

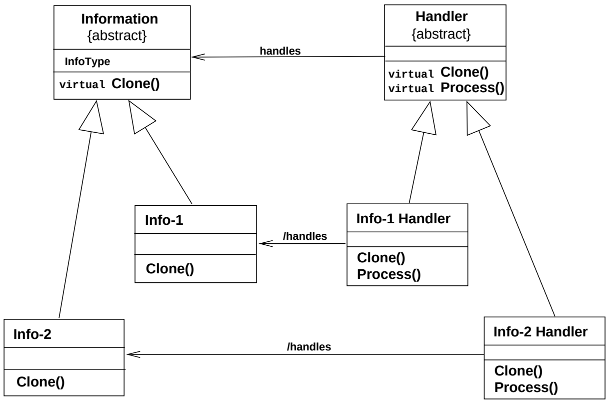

Figure 2. Extract of the data model for transferable information objects and their handlers in the UML notation [10]. The symbol '/' at the lower 'handles' associations indicates their inheritance relationship to the corresponding upper association.

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* [11, pages 117ff]. The idea behind this design pattern is that different classes of information objects and their handlers are represented through 'prototypical' instances that are able to 'clone' themselves. Figure 2 displays the class structure for the prototypes of information objects and their corresponding handlers. Handlers process received information. Figure 3 illustrates the architecture of the communication framework which only needs to know the abstract classes Operation and Handler, and not their concrete subclasses.

A more detailed explanation of the models in Figures 2 and 3 is given as follows. Rectangles are the UML symbols for classes. Within a class rectangle, the class name (at the top), attributes (in the middle) and methods (at the bottom) are defined. The C++ keyword `virtual` [28] is used to specify abstract methods in abstract classes. Inheritance for specialization and generalization is shown in UML as a solid-line path from the subclass to the superclass, with a hollow triangle at the end of the path where it meets the superclass [10]. Using UML, multiplicities for associations are specified through numerical ranges at the association links (Figure 3). The default multiplicity is 1. If the multiplicity specification comprises a single star, then

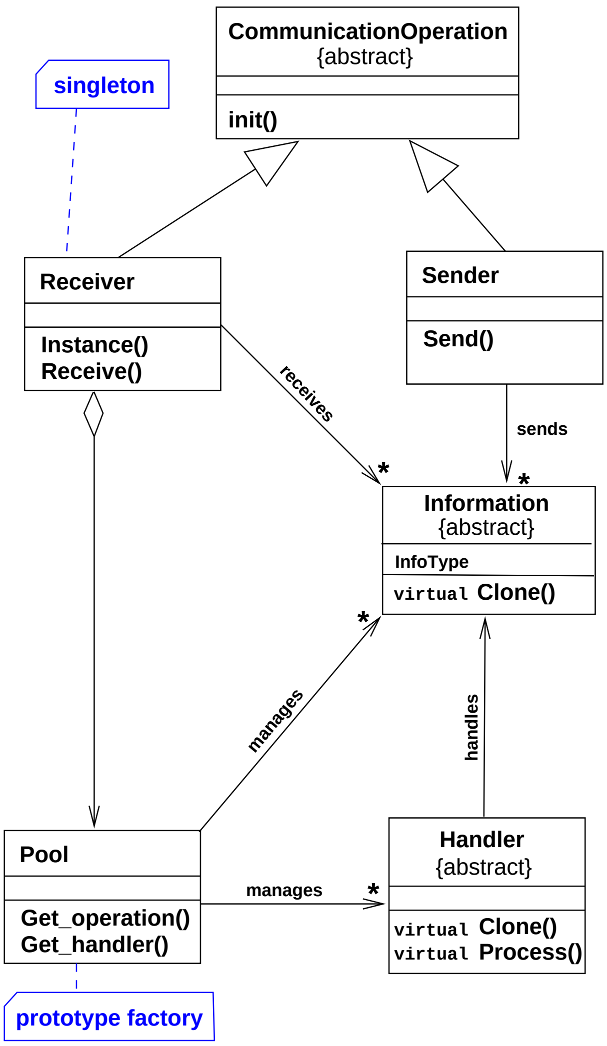

Figure 3. The general architecture of the previous C++ communication framework.

it denotes the unlimited non-negative integer range (zero or more). Hollow diamonds indicate part-of relations (aggregation). The arrows indicate the access direction. The applied design patterns *Singleton* and *Prototype Factory* are indicated through comment boxes that are attached to the corresponding classes via dashed lines in Figure 3.

The classes Sender and Receiver manage the transfer of information. They inherit some general methods for using the Object Request Broker from the abstract class CommunicationOperation (see Figure 3). The communication framework uses two abstract classes for which a user specifies concrete subclasses (see Figure 2):

Information: for each type of information a concrete class is defined through inheritance from the abstract class Information which specifies a uniform interface for all information types (see Figure 2). Each concrete subclass specifies the specific structure for the specifications of one type of information to be exchanged via object instances of this class. The communication framework itself is independent of this specific structure.

The Clone method is needed to obtain copies of the prototype objects. The attribute InfoType identifies the type of the prototype objects.

Handler: to receive and process information of a specific type, it is necessary to provide corresponding information handlers to process the information in an appropriate way (see Figure 2). On receipt of an information object, the communication framework uses copies of *prototype* objects for information/handler pairs, which are managed by the class Pool (see Figure 3). The handler is responsible for processing the associated information; thus, realizing the corresponding application logic.

The presented mechanism, which makes the communication framework independent of the concrete information classes, has been achieved through guiding the design by the pattern *Prototype Factory*.

Another design pattern used in Figure 3 is called *Singleton* [11, pages 127ff]. Each CORBA object (a C++ program) contains exactly one C++ object instance of the class Receiver, because each database agent is accessed as a CORBA object. However, several Sender objects may exist within a CORBA object.

It turns out that the developed communication framework is an object-oriented framework with *inversion of control* [9]: 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.

With the presented 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 decoupling 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.

We can only present an coarse overview of this architecture in this paper. For a more detailed description refer to [14]. The Chorus Cool CORBA implementation [18] was deployed in this project.

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 3) 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 4, 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.

We have attempted to combine these ideas with previous research undertaken on metadata modeling in multi-database systems [26]. In this research one of the functions of the canonical model was to represent the *virtual* schemas (export and federated schemas [27]) 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 [7] object types, and clones can be constructed from these prototypes to carry data values.

3. Generative Communication in Linda

There has been particular attention on parallel computing within the computer science community in the last decades. 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:

> "In fact, even though PVM and the MPI [8] are de facto standards in parallel programming, their related programming style looks in many respects like assembler-level programming of sequential computers." [29]

In particular, the lack of a global name space forces algorithms to be specified at a relatively low level, since it is complicated to simulate shared memory [3]. This greatly increases the complexity of programs, and also restricts algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions. Therefore, several alternative models have been designed for parallel programming, which provide higher-level abstractions. These languages emphasize some kind of shared data.

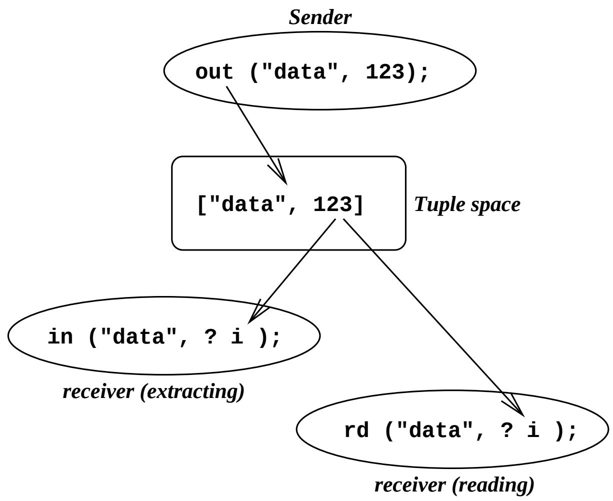

Figure 4. Tuple-space communication in Linda. Processes are displayed as ellipses and the tuple space is displayed as a rectangle.

One of these languages is Linda that has been developed by Nick Carriero and David Gelernter at Yale University [4, 12]. The shared data pool in the Linda concept is called *tuple space* which is a collection of tuples. A tuple space may contain any number of copies of the same tuple: it is a multiset, not a set. 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 [12]. Figure 4 illustrates this concept which is explained below.

Reading access to tuples in tuple space is associative and not based on physical addresses. Reading access to tuples is based on their expected content described in so-called *templates*. This method is similar to the value-based selection of entries from a relational database. Each component of a tuple or template is either an *actual*, i.e., holding a value of a given type, or a *formal*, i.e., a declared placeholder for such a value. A formal is prefixed with a question mark. 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 "`out("data", 123);`" operation in Figure 4 deposits the tuple `["data",123]` into 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*: a tuple can be grabbed by only one process, and once grabbed it is withdrawn entirely. 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 "`in("data",?i);`" operation in Fig. 4 withdraws the tuple `["data",123]`, which matches the template "`["data",?i]`", from tuple space and assigns `123` to the integer variable `i`. To summarize, a tuple and a template match in Linda iff

- the numbers of fields are equal,
- types and values of actuals in templates are equal to the corresponding tuple fields, and
- the types of the variables in the formals are equal to the types of the corresponding tuple fields.

The `rd` operation is the same as `in`, with actuals assigned to formals as before, *except* that the matched tuple remains in tuple space (Figure 4). Additionally, many Linda dialects provide non-blocking extraction operations, multiple tuple spaces and support for process creation. Some Linda dialects allow formals in deposited tuples which match with appropriate actuals in templates. Refer to [5] for a full account to parallel programming in Linda. Comparisons of Linda with other approaches to parallel programming may be found in [4, 21]

A *coordination language* like Linda provides means for process creation and inter-process communication which may be combined with *computation languages* like C [6]. A *parallel programming language* consists, therefore, of a coordination language and a sequential computation language. With Linda, coordination and computation are two separate issues of equal standing which together address the problem of building software. The first computation language, in which Linda has been integrated, was C. Meanwhile there exist also integrations into higher-level languages supporting the early phases in software development, such as prototyping for early design evaluation [16].

Parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to focus on more than one process at a time. Linda's shared, associative object memory supports a highly *decoupled* programming style in which processes remain mutually anonymous. Each task in the computation can be programmed (more-or-less) independently of any other task. 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 exists at the same time and do not have to do things synchronously.

A flexible coordination device is the distributed data structure which is well-developed in the Linda programming model of generative communication [20]. Distributed data structures are data structures that can be manipulated simultaneously by several processes. Processes communicating via distributed data structures do so with minimal coordination: processes may deposit data without being aware of the receivers who will access it. Processes may access data without being aware of the producers who generated it. This implies asynchronous behavior, since the generation of information is decoupled from its consumption.

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.

4. 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 (Section 2) and generative communication (Section 3). 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, their 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 the object type has been registered previously with the GC service. If not, it must 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:

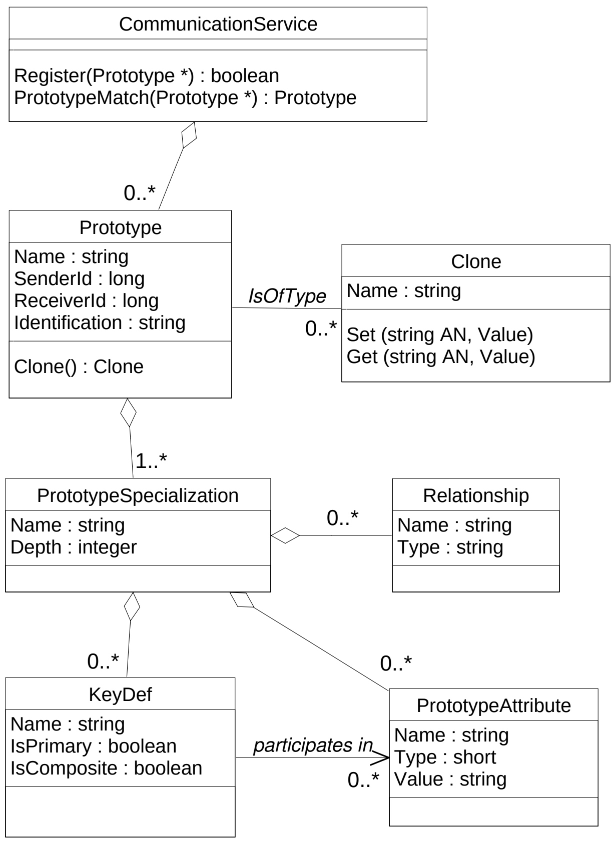

Figure 5. The metamodel of the Generative Communication Service modeled in the UML notation for class diagrams [10]. Hollow diamonds indicate part-of relations (aggregation).

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* model 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. Our work differs from the framework discussed in Section 2 in that although our prototype factory handles objects of unknown types, it *understands* how to represent them through storage of the objects' metadata. This provides us with a useful matching technique for object prototypes which we will discuss later in this section.

4.1. Metamodel Description

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 generic class, which describes new types discovered at runtime, and contains some 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 3 where formal parameters are metadata elements and actual parameters are data elements. Our metamodel which is illustrated in Figure 5, has two data elements (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

```
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 6. Extract of the ODL interfaces for clones and prototypes.

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.

4.2. 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 [22]. 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.

The ORB is the middleware that establishes the client-server relationships between objects. Clients can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can accept the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. Server objects offer services to client objects.

Our architecture assumes a federation of databases 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 [27]) 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 any object which looks like this?'. 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 in 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 SQL-type query can refine this dataset to those objects which match a certain criteria. This way not all objects may be downloaded.)

Unlike Linda, the matching of more than one prototype does not result in an arbitrary selection of one of them. Instead the `identification` attribute in the `Prototype` class is used 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 [7]) help to illustrate the differences between prototype and clone objects in the implementation (Figures 6–8). 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 among them, as displayed in Figure 6. 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 all `Clone` objects look similar 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` 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 6 and 7).

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.

```
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 7. ODL interfaces for specializations.

The final ODL sample in Figure 8 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 and the actual data type. The `AttributeType` attribute in the `PrototypeAttribute` class contains an enumerated type denoting system datatypes, a relationship, a relationship set or an ODMG collection.

4.3. 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 9 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

```
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 8. ODL interfaces for attributes.

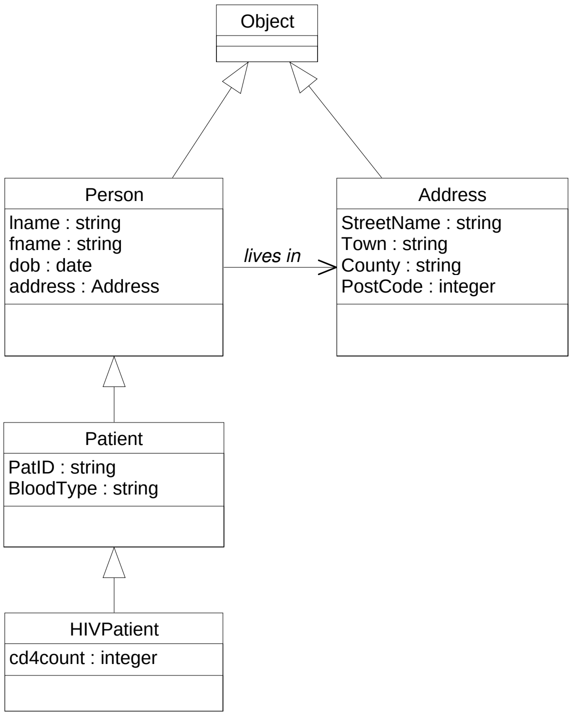

Figure 9. A sample healthcare object to be passed using the Generative Communication service.

be created for the `Person` prototype unless the `Address` prototype is also registered.

The registration process first requests a class 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 the 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.

4.4. Synchronization

The CORBA event services are used in combination with our GC service to provide synchronous or asynchronous

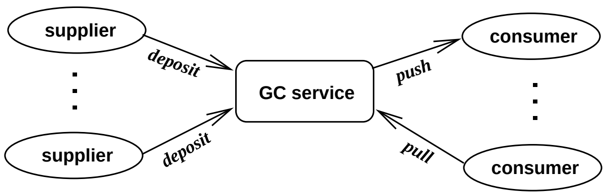

Figure 10. Push and pull communication through the GC service.

transfer of information objects using event channels [23]. 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 10 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 [17], and the result is a flexible coordination architecture.

4.5. 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 which is part of a future revision where the GC service provides a table of public keys for each potential consumer. It means 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.

5. Summary and Future Work

This paper starts with a discussion of previous and related work on object-oriented communication frameworks (Section 2) and generative communication in parallel programming (Section 3), before 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 quality properties that should be achieved in system's design [13]. One output is that the GC service can be re-used for other systems with similar communication requirements, in particular exchange of information among cooperative information systems.

Our current and 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 which we handled a situation where a consumer's query results in more that 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. 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

This work was 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.

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] H. Bal. *Programming Distributed Systems*. Silicon Press, 1990.

[4] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[5] N. Carriero and D. Gelernter. *How to write parallel programs*. MIT Press, Cambridge, MA, 1990.

[6] N. Carriero and D. Gelernter. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, Feb. 1992.

[7] R. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufman, San Francisco, CA, 1996.

[8] J. Dongarra, S. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Commun. ACM*, 39(7):84–90, July 1996.

[9] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997.

[10] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, Reading, MA, 1997.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

[12] D. Gelernter. Generative communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, Jan. 1985.

[13] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[14] W. Hasselbring. Federated integration of replicated information within hospitals. *International Journal on Digital Libraries*, 1(3):192–208, Nov. 1997.

[15] 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. (in press).

[16] W. Hasselbring. The ProSet-Linda approach to prototyping parallel systems. *The Journal of Systems and Software*, 1998. (in press).

[17] IONA Technoligies PLC., Dublin, Ireland. *White Paper – OrbixTalk*, 1997.

[18] C. Jacquemot, P. S. Jensen, and S. Carrez. CHORUS/COOL: CHORUS object oriented technology. In *Object-Based Parallel and Distributed Computation (OBPDC '95)*, volume 1107 of *Lecture Notes in Computer Science*, pages 187–204. Springer-Verlag, 1995.

[19] R. Johnson. Documenting frameworks using patterns. In *Proc. OOPSLA '92*, pages 63–76, Vancouver, BC, Oct. 1992.

[20] M. Kaashoek, H. Bal, and A. Tanenbaum. Experience with the distributed data structure paradigm in Linda. In *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 175–191, Ft. Lauderdale, FL, Oct. 1989.

[21] A. Matrone, P. Schiano, and V. Puoti. Linda and PVM: A comparison between two environments for parallel programming. *Parallel Computing*, 19(8):949–957, Aug. 1993.

[22] T. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley, New York, 1995.

[23] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Object Survival Guide*. Wiley, New York, 1996.

[24] E. Pitoura, O. Bukhres, and A. Elmagarmid. Object orientation in multidatabase systems. *ACM Comput. Surv.*, 27(2):141–195, June 1995.

[25] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Wokingham, England, 1995.

[26] 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.

[27] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[28] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.

[29] D. Talia. Parallel computation still not ready for the mainstream. *Commun. ACM*, 40(7):98–99, July 1997.

[30] J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, 1996.