Report

Principles of distributed object database languages

Author(s):
Tresch, Markus

Publication Date:
1996

Permanent Link:
https://doi.org/10.3929/ethz-a-006651652

Rights / License:
In Copyright - Non-Commercial Use Permitted
Principles of Distributed Object Database Languages

Markus Tresch

Institute for Information Systems
Swiss Federal Institute of Technology (ETH)
CH - 8092 Zurich, Switzerland
Email tresch@inf.ethz.ch

Abstract

Today's information systems are based on an architecture built up of object servers and client applications that are distributed in a network of multiple heterogeneous computing platforms. In this environment, distributed object-database systems provide efficient and reliable storing, querying, and updating of a large diversity of objects.

The concern of this paper are database languages for uniform querying and updating of distributed objects. The fundamental principles of languages for distributed object database systems are presented, such that based on that, languages for centralised object databases can be extended for managing distribution. Five different distribution architectures with decreasing degree of local autonomy and increasing degree of global control are presented, for each of which the power of the distributed object database language is compared.

1 Introduction

Management of distributed objects is one of the most important recent developments in information technology research. The promise is that this technology will finally facilitate global (top-down) design of distributed data management systems, as well as (bottom-up) integration of distributed legacy applications. In this framework, distributed object databases play a key role since their objective is the transparent and efficient management of objects that are persistently stored across multiple heterogeneous sites.

From a research point of view, distributed object databases can best be characterised as the merger of technologies from the three main research areas distributed databases, object-oriented databases and distributed object management. Distributed databases consider the traditional (relational) issues, primarily physical allocation of replicated and partitioned data [OV91]; data localisation and monitoring to improve performance of distributed queries, e.g. by intra- and inter-query parallelism; and distributed transaction management to make them reliably in a distributed environment. Object-oriented databases deal with the efficient persistent management of large collections of complex objects. Main issues are object-oriented data modelling; languages for querying and updating objects; the physical storage of objects optimised for fast retrieval (clustering, indexing); and advanced transaction processing satisfying non-standard application requirements. Distributed object management is focusing on mechanisms for network communication between objects.
(cf. CORBA standard [COR95]) and application architectures for sharing objects in common servers (e.g., object paging and caching strategies) [MHG+92, OV94, ODV94]; and mechanisms for managing objects in large scale distributed object systems.

From a system technology point of view, objects in distributed object database systems are distributed over multiple sites, at each of which object database server and/or application client software is running. As illustrated in Figure 1, there are several reasons why objects are distributed:

- **Object Caching:** Objects are persistently stored in database servers. They are used in application client programs that may run at the same or at a different site. Consider for example the white object stored on the server of site C and used by two clients, one at remote site A and one at local site C. Application clients may either request a copy of the server object for local caching and processing [FC94], or they just hold the object identifier and request methods to be processed at the server.

- **Object Replication:** Objects may be replicated over multiple servers for safety or performance reasons. Consider again the white object that is replicated at servers of sites C and D. Application client C is using the local replication. However, this might be completely transparent to the client, such that e.g., updates to replication at C are automatically propagated to site D and consistency of replications is maintained by the system [JM90].

- **Object Fragmentation:** Again for safety and performance reasons, objects may be fragmented over multiple servers [KNM94]. For example, the gray and the black object, both are fragmented. Client application at site B used all information of the black object by collecting data from local site B and remote D. In contrast, client A is using the gray object of local server A. The fragmentation of this object is transparent to client A, because the database server at site A offers an integrated view of the partitioned gray object.

![Figure 1: Distributed Object Database System](image)
The concern of this paper are languages for querying and updating such objects that are distributed over multiple database servers. Whereas centralised objects systems are pretty well understood, distributed object technology needs intensive further research. With respect to object database languages for example, a variety of proposals have been made in the past for the management of centralised object systems. Many approaches extended the well formalised relational algebra towards an object algebra, like for instance [ASL89, SZ89, SÖ90, SLR+92], which lead to the proposition of object-extensions to SQL, that allow for select-from-where-like queries on object-oriented databases, like for instance the O2's query language [BDK91], and recently to the standard ODMG-93 [Cat94].

Enhancing these languages in order to manage distributed objects is not straightforward and raises open questions. For example, the data model is object-oriented, such that there is a need for global object identification mechanisms for distributed objects. Distributed type systems must be combined into a global type lattice enabling static and dynamic type checking of distributed language expressions. Relationships and integrity constraints among objects of different databases must be respected by distributed queries and updates. And further object-oriented concepts, like method inheritance, overriding, and data encapsulation may span distributed object databases.

In this paper, we elaborate on the principles of languages for distributed object databases. Once these distribution principles are known, different related systems can be compared, and languages, methods, and tools can be built on top. The contribution of this paper is twofold:

1. Languages for centralised object databases can be extended for the management of distributed objects by implementing well defined distribution principles discussed in this paper. Hence, it is not our intention to present yet another data model or language, but to identify and formalise the fundamental concepts of distributed object database languages. This allows for sorting out terminology, language implementations, architectural implications, as well as standardisation efforts.

2. The underlying distributed object database architecture, or more precisely their implicit degree of autonomy, determines the implementation of the above principles and hence the power of the distributed object database language. Five different distribution architectures with decreasing degree of local autonomy and increasing degree of global control are presented, for each of which the distributed object database language is sketched and compared with each other.

It turns out that many important issues that are usually raised while discussing distributed object databases are orthogonal to the distribution principles presented in this paper. Among them are transformation of heterogeneous data models into a canonical representation, schema integration methods, object replication/allocation schemes, distributed transaction processing, or distributed communication infrastructures. Since these issues do not affect the results in this paper, we intentionally do not consider them, they are out of the paper scope. Other distribution issues are directly related and principles of distributed object databases languages can not be discussed independently, like for instance the choice of architecture, as we will show later.

This paper is organised as follows. In Section 2, we discuss five different architectures of distributed object database systems with different levels of local autonomy. In Section 3, we
identify the fundamental principles of distributed object database languages. In Section 4, we show how these principles are implemented using the distributed object database language COOL* as an example, that is scalable to any architectural level presented before. Section 5 discusses related work and Section 6 gives an outlook on how the results of this paper are currently implemented in a prototype system.

2 Architecture of Distributed Object Databases

The term distributed object database does not provide any information about the underlying distribution architecture. Figure 2 illustrates three main architectures of distributed object databases: (a) multi-database systems, (b) federated database systems, and (c) physically distributed database systems.

![Figure 2: Architectures of Distributed Object Database Systems](image)

The difference between these architectures is how tight the object database servers are cooperating. At the left end of the scale, multi-databases describe very loosely coupled object database systems, and at the right end, physically distributed database systems are very tightly coupled. In between are the federated databases [HM85, SL90].

Based on this notion of loosely and tightly coupled object database systems, we introduce a further refinement and classify distributed object databases architectures into five formal levels of database cooperation:

- Level 0: Multi-Database Systems,
- Level I: Composition of Federated Object Database Systems,
- Level II: Virtual Integration of Federated Object Database Systems,
- Level III: Real Integrated Federated Object Database Systems, and
- Level IV: Physically Distributed Object Database Systems.
These levels describe a scale with increasing degree of global control and decreasing degree of local autonomy, and thereby characterise this fundamental tradeoff. In the sequel, we briefly introduce each level.

### 2.1 Multi-Database Systems (Level 0)

Multi-database systems refer to the most loosely coupled architecture, where object servers are not integrated at all. Databases are fully autonomous and not even known from each other. Database schemas are translated into a common representation, the joint data model. As illustrated in Figure 2a, multi-database systems provide no global schema and there is no globally merged name space. Consequently, schema integration is not available to resolve semantic conflicts [BLN86] and objects from different databases are always disjoint.

Since a global schema is not possible at level 0, database integration is completely up to the application program, which can be seen as ad hoc "integration". Interoperability must be hand-coded by writing query and update statements involving objects from multiple databases. Distributed transaction management might be employed to process objects from different databases within one global transaction and therefore to achieve some form of distributed processing.

### 2.2 Federated Object Database Systems (Level I - III)

Federated database systems lie in between two extremes (cf. Figure 2b). Their objects and schemas are subject to some global control, while at the same time, participating servers have retained some autonomy. Global schemas exist integrating information from multiple databases. Application programs may either use these global schemas or may operate on a local database without knowledge of the global system. There exists a *distribution dictionary* that is used to store global information about distribution, integration, and transformation of objects and schemas.

Federated object databases are the most challenging architecture and are further distinguished into levels I, II, and III.

#### Composition of Federated Object Database Systems

Integration level I is the "weakest" form of federated databases, where participating systems are willing to give up some local autonomy to cooperate but nevertheless keep a very high degree of autonomy. Autonomy is restricted because databases have to export parts of their schema to the federation and therefore have to notify the federation system if they change the schema that is exported to the federation.

Minimal database integration is available only, that we call *schema composition*. It produces a global database by putting schemas side-by-side and merging name spaces. Schema composition is the first step towards closer system cooperation, while at the same time preserving maximum autonomy. No further links between the databases are established. In particular, the global schema can not be changed or augmented.

Schema composition composes a global meta database too. As explained earlier, we do not consider data model heterogeneity in this paper, but we assume that all databases are already translated into the same data model and furthermore, meta schemas of all
databases are identical. Schema composition now combines these meta databases. Refer to [SST94] for an in-depth discussion of combining meta schemas of multiple databases.

Application programs using the global composed schema may invoke global operations that include names of object and schema elements from different databases within one statement.

Virtual Integration of Federated Object Database Systems

Based on the global schema created by schema composition of the previous level, integration level II now allows for the definition of further links between objects and schemas. The restriction of this level is that these links are strictly virtual, i.e., they are given by a query expression or a view definition. Distributed queries and updates can be written using these multi-database views.

The definition of these virtual links (views) are stored as meta-information in the distribution dictionary. No other information than meta-data can be stored in the distribution dictionary. For example, one can not use it to store objects. This would require further loss of autonomy, that is not allowed at level II.

The result is a virtually integrated global schema with additional transparency for the global user. Object integration, schema integration and changes (conflict resolution, unification, restructuring) can be performed as long as they can be expressed using views. The state of the global database is still strictly virtual, that is, it can be derived from the state of the underlying databases at any time.

Real Integration of Federated Object Database Systems

At level III, the limitation of using virtual integration (view techniques) only is removed. Objects and schemas can be integrated by establishing real (stored links) between databases. As a consequence, the schema can be further integrated and can even be extended by global concepts.

The use of the distribution dictionary is extended in order to store and manage not only meta-information, but database objects too. This is an important extension, causing further loss of local database autonomy. Now, databases have to export the local object identifier (OID) to the federation that must be able to store these OIDs in the distribution dictionary. Furthermore, (some) objects may partially be copied in a global database. Consequently, federated object databases of level III now have a real global state.

Schema integration and augmentation is not limited to views any more. In contrast, global schema elements can be added in order to integrated or reorganise the global schema.

2.3 Physically Distributed Object Database Systems (Level IV)

We refer to the most tightly coupled systems as level IV, where only one management system exists as physically distributed database system. Hence, “distributed” simply refers to the physical distribution of objects over multiple storage systems (cf. Figure 2c). Participating database systems completely lost their autonomy. Though objects might be physically distributed, these systems have one single logical database schema. Distribution is therefore a purely physical issue and is logically transparent.
3 Distribution Principles

In this section, we identify the fundamental principles of distributed object database languages. As mentioned before, these principles are valid independent of a particular database language or type of language. They apply for extending any object-database language towards managing distributed object databases, like for instance an object algebra, an object-extension to SQL, or the ODMG-93 language.

To proof this fact, we use in this paper the object algebra COOL [SLR+92] and extend it to the distributed object database language COOL* [Tre95]. COOL distinguishes itself from other above mentioned approaches, in that model and language are formally well defined, and that it includes a set of generic update operations [LS92]. The later is crucial for our purposes, because understanding of many distribution principles can only be achieved by understanding the effect of updates in distributed object database languages.

Concrete realization of the presented distribution principles highly depends on the distributed object database architecture, introduced in the previous Section. For the time being, we assume a level III architecture (real integration of federated object database systems), hence, databases are generally open for cooperation with other databases, but are still autonomous, such that cooperation must not affect local database operations. We come back to this issue in detail in Section 4.

Example and Notation

Throughout this paper, a travel information system which is distributed over three different object database sites is used as running example: database FlightDB (DB1) holds airline and flight information, database CityDB (DB2) holds city tourist information, and database TravelDB (DB3) holds the travel office booking data. As illustrated in Figure 3, these three databases are mainly modelled as classes (shown as ovals) of objects with attributes. Sub-classing is illustrated using thick arrows and relationships (functions) between objects are given as thin arrows.

Figure 3: Distributed Travel Information System
If schema element names are globally not unique, we qualify them by the (short-)name of the database using @-notation, as for example Objects@DB1 or name@CityDB. In Figure 3, databases are semantically related to each other in many ways. This is indicated by (thin and thick) arrows spanning different databases. The explanation of the semantics of these inter-database connections is the main concern of the remainder in this section.

3.1 Compatibility of Data Types

Data types model the primitive (printable) values of object database systems, like for example, integer, real, boolean, or string. A most fundamental assumption is that these data types are identical within all object databases, or at least, that there exist equivalence preserving mappings. It requires for example, that integers are internally represented with the same storage size and that real numbers have the same precision.

Compatibility of data types is the precondition for comparing values of different databases with each other. In Figure 3 for example,

\[ \text{name@DB1}(d) = \text{name@DB2}(c) \]

compares the name of destination \( d \) in FlightDB with the name of city \( c \) in CityDB.

3.2 Distributed Object Identification

In distributed object databases, entity objects determine real world objects in the distributed system as a whole and proxy objects represent the real world object in a particular database [Ken91]. This essential distinction allows for modelling the fact that in distributed database systems one entity object can be represented in multiple object databases at the same time.

Hence, the fundamental assumption of object-oriented systems, that one real world entity is represented by one database proxy, does not apply. Consider for example Figure 3, the same city may be represented as an object of class Cities in CityDB and of class Destinations in FlightDB. This leads to the major question of how to globally identify entity objects in distributed object databases.

Object identifiers (OIDs) serve as the technology for logically identifying proxy objects within one database. In a distributed system, OIDs can be assigned to objects by a centralised OID server or decentralised by each databases itself. For our purposes, centralised OID assignment is unsuitable because it restricts local design autonomy. Furthermore, we assume that some object databases may be legacy systems, hence becoming part of the distributed database after local objects have been created.

Consequently, OID domains of local databases are pairwise disjoint, and the concept of object identifiers is inadequate for global identification. In contrast, entity objects can exclusively be identified by characterising values (called value identifiability in [Bee93]), a generalisation of what we know as keys from relational databases.

However, OIDs determine local object identity, such that two objects can only be equal if they stem from the same database. This is obviously not satisfactory and raises the question of how to model that two proxy objects \( o \) and \( o' \) represent the same real world entity. We compare an object generating and an object preserving approach:
- **Object generating.** A new object \( o'' \) is generated in one or both databases as an integrated copy of \( o \) and \( o' \). \( o \) and \( o' \) are either deleted or \( o'' \) could be treated as a replication. A variant is that one object is integrated into the other. In any case, this approach is a database update and one must make sure that references from and to objects \( o \) and \( o' \) are handled correctly.

If \( o'' \) is created with the same OID in both databases, it violates the above mentioned local autonomy, since this OID must be assigned by a global OID-server, and is therefore only possible for “tightly-coupled” distributed object databases (see Section 2). If \( o'' \) is created as an integrated object in one database only, it requires migrating the object from the other database, which violates the requirements that local operations shall not be affected.

- **Object preserving:** No new objects are generated, but existing objects \( o \) and \( o' \) are put in relationship with each other. This is not an update operation, but can be seen as an object preserving join operation. Depending on the data model, it can be accomplished by defining inter-database relationships, referencing an object in one database with an object in another database. These references can either be stored or derived.

The object preserving way of integration approach is compatible with the local autonomy requirements as well as the legacy database issues. Furthermore, it does not impose any restrictions to the tightness of local database cooperation. We pursue this last approach.

We use a concept that has been introduced in [SST94, TS94] (see also Section 5) and say that two proxy objects \( o \) of database 1 and \( o' \) of database 2 are the same if they represent the same real world entity object. Formally, we define partial, injective, single-valued functions, which in the sequel will always be called \( \text{same}_{i,j} \):

\[
\text{function} \ \text{same}_{i,j} : \text{object@DB}_i \rightarrow \text{object@DB}_j
\]

These \( \text{same} \)-functions have a special semantics, returning for a given DB\(_i\)-proxy object "the same" proxy object in DB\(_j\). The use of \( \text{same} \)-functions is illustrated in Figure 3 where such an inter-database function is defined from \( \text{Destinations@DB1} \) to \( \text{Cities@DB2} \).

In the stored variant, \( \text{same} \)-functions are realized as a relation (table) with tuples \( <o, o'> \), grouping objects together that represent the same real world object. Notice, that this relation is stored in the distribution dictionary. In the derived variant, \( \text{same} \)-functions are methods using a query language for example, to compute the same object. The power of the query language determines what kind of objects can be integrated. This is not a database update. The method implementation code needs to be stored in the global distribution dictionary.

Notice that in general, \( \text{same} \)-functions can not be generated automatically, but must be defined by the database administrator, maybe supported by tools. Using \( \text{same} \)-functions, we are now ready to define distributed object identity:¹

¹The notation \( \text{object}(o) \) is to be read as a type predicate which is true iff \( o \) is an instance of type \( \text{object@DB} \), and therefore stems from database DB\(_1\).
Definition 1. (Distributed object identity) Distributed object identity \( (=_g \) of two objects \( o, o' \) is defined as

\[
=_g : \text{object} \times \text{object} \rightarrow \text{boolean}
\]

\[
o =_g o' \iff (\exists i : \text{object}_i(o) \land \text{object}_i(o') \land o =_i o')
\]

\[
\lor (\exists \text{same}_{i,j} : \text{object}_i \rightarrow \text{object}_j : \text{object}_i(o) \land \text{object}_j(o') \land o' =_j \text{same}_{i,j}(o)).
\]

From now on, two objects are the same, if they stem from the same database and are identical in it, or if they have been integrated (by the user/DBA) using \( \text{same} \)-functions. This special semantics of \( \text{same} \)-functions – or of any equivalent mechanisms in any other language – is fundamental for distributed object database languages. Query and update operations must be made aware of this special semantics and must respect its semantics.

This includes that static and dynamic type checking must respect this distributed object identity based on \( \text{same} \)-functions too. It requires mainly the application of the following type inference rule for two objects \( o \) and \( o' \) from different databases

\[
o : \tau, \ o' : \tau', \ o' = \text{same}(o)
\]

which says that if \( o \) is of type \( \tau \) and \( o' \) is of type \( \tau' \) and there exists a \( \text{same} \)-function between these two objects, then \( o \) is also of type \( \tau \cap \tau' \), that is, the common subtype of \( \tau \) and \( \tau' \).

3.3 Inter-Database Object Relationships

Functions are used to model type-specific properties of objects (the function’s domain is always an object type). If the range type of a function is a data type, the function models an attribute, if it is an object type, the function models a relationship. Functions are always partial, their value may be undefined. In Figure 3 for example, attribute \text{name} \ and \text{relationships} \text{op_by} \ and \text{operates} \ are defined as follows

\[
\text{function name} : \text{airline} \rightarrow \text{string};
\]

\[
\text{function op_by} : \text{flight} \rightarrow \text{airline} \quad \text{inverse operates};
\]

\[
\text{function operates} : \text{airline} \rightarrow \text{set of} \text{flight} \quad \text{inverse op_by}
\]

Two functions can be inverse to each other, implementing an integrity constraint. Sets are used to collect objects/values together. For example, functions may be set-valued, returning a set of objects/values, as shown in the function \text{operates}.

Functions are not only stored but can also be derived (computed) access methods, where the implementation of the function is given by a query expression (see Section 4). We consider side-effect free functions only, that is, functions not updating the database. For example, function \text{flight_time} \ computes the flight time from stored arrival and departure values:

\[
\text{function flight_time}(f) : \text{flight} \rightarrow \text{real}
\]

\[
\text{as} \text{arrival}(f) - \text{departure}(f)
\]

Domain and range type of functions may be of different object databases, hence spanning multiple databases. The following functions for example are defined at object database DB3, hence forming an extension of the DB3 schema:
At DB3:  

function booked_flights : booking \rightarrow \text{set of flight} @ DB1;

function booked_US_flights : booking \rightarrow \text{set of flight} @ DB1 
  as b: \text{select}(\text{country}(to(f)) = "US") \{f: \text{booked_flights}(b)\}

Both are inter-database functions, defined on DB3 objects and returning sets of DB1 objects. Function booked_US_flights is a derived inter-database function. It selects those flights heading for the US. This computation involves functions to@DB1, country@DB2, and booked_flights@DB3 from three databases. In case of stored functions, their actual values are stored in the distribution dictionary, in case of derived functions, their defining query expressions are stored in the distribution dictionary.

Notice that same-functions that we introduced earlier are regular inter-database functions, but with some additional semantics, defined in Definition 1.

### 3.4 Inheritance over Distributed Databases

Abstract object types model database objects by describing their common interface as a set of applicable functions. For example, object type employee can be defined as inheriting all functions from type person and having additional functions salary, dept, and manager:

\begin{verbatim}
  type employee isa person = salary, dept, manager
\end{verbatim}

The sub-/super-type relationship between object types (also called inheritance or type extension) is defined by the subset of interface functions. Notice that this notion of a subtype relationship is rather implicit and forms a type lattice.

The type lattice of every database DB_i has a root which is the most general object type object@DB_i. All objects of DB_i are at least instance of this type. object@DB_i is defined as the type with only one function — object identity (=) — for comparing whether two objects are identical. The object type bottom@DB_i is the most special object type, being the common subtype of all object types and having all functions as its interface.

To enable type checking over multiple databases and function inheritance from another database, type systems must be merged. A global root type object@GDB is introduced, that is the common upper-bound of the global type lattice of multiple object databases. The interface of type object@GDB defines one function, the distributed identity function =_g as it was introduced in Definition 1. The common lower-bound bottom@GDB is the global subtype with all functions of multiple databases as interface.

We do not require the merging of all type systems, but allow an incremental approach of the involved databases only. Figure 4 sketches the resulting global object type lattice of a distributed database consisting of two databases. \([f_1, \ldots, f_n]_i\) denotes an object type with functions \(f_1, \ldots, f_n\) in DB_i. As a consequence of the type system being a lattice, such types exist even if they have not explicitly been defined and are therefore unnamed.

The approach of having implicit subtype detection based on the extension of type interfaces is also known as the concept of conformity [BHJL86]. Others, like for example [CHS+95], achieved later through this principle additional flexibility when independently defined schemas are merged.

Bold types and inheritance arrows in Figure 4 form types that are established after creating root type object@GDB. The instances of these types are allocated over multiple
Figure 4: Distributed Object Type Lattice

databases. Local type lattices remain unchanged, but new global object types are created whose interfaces consist of functions from both databases.

Given a merged type lattice, object types may now be defined that span multiple databases. Consider

At DB3:  type employee isa person@DB1 = salary@DB2, dept@DB2, manager

Type employee is defined in DB3 as a subtype of person@DB1, extended with functions from local database and database DB2. According to Figure 4, this type already existed, but had not been given a name. Instances of this type are distributed over databases DB1, DB2, and DB3. In Section 4, we will discuss the preconditions for creating such instances.

3.5 Overriding and Data Encapsulation

Overriding is common in object-oriented languages. It is important in distributed object database systems, because it provides the possibility to locally re-implement functions from another database. To illustrate overriding in distributed databases, we introduce a function overriding scheme, where functions can be re-implemented at a subtype. The domain of the function must remain unchanged. Stored as well as derived functions can be overridden by stored or derived functions.

In the following example, type employee@DB3 is defined as subtype of person@DB1 and locally overrides function age:

At DB1:  type person = name, age;
          function name : person → string;
          function age : person → integer
                          as ... some method code ...

At DB3:  type employee isa person@DB1 = manager;
          override age : employee → integer
                          as ... some different method code ...

12
As a consequence of this overriding mechanism, polymorphism and late binding apply as known from any object-oriented language, but extend to multiple databases. Consider two objects of the following types as an example of a polymorphism involving multiple databases:

At DB3:

```plaintext
var p, x: person@DB1;
var e: employee;
...
if P(...) then x := p else x := e;
print age(x)
```

Depending on the current value of P, the implementation of `age` of DB1 or DB3 is called, which requires a late binding mechanism.

Overriding together with polymorphism (which requires late binding) achieves a form of transparency in distributed database systems, that determines at run-time, whether the local or remote function implementation is called.\(^2\)

Data encapsulation is another very common object-oriented mechanism, allowing for achieving a high level of transparency in distributed database systems. To realize data encapsulation, any kind of function (stored/derived, attribute/relationship) can be marked as private, hence being accessible for own functions only (cf. e.g. C++ class definitions). In the following example, function `booked_flights` is not directly accessible, but is used by public function `booked_US_flights`:

At DB3:

```plaintext
type booking = booked_US_flights, private booked_flights;
function booked_flights : booking@DB3 → set of flight@DB1;
function booked_US_flights(b) : booking@DB3 → set of flight@DB1
    as select [country(to(f)) = "US"] (f: booked_flights(b))
```

Thanks to data encapsulation, it is transparent to the user of database DB3, that the computation of the result he requested needs data from multiple other databases.

### 3.6 Global Classes and Variables

Classes are strictly separated from object types and represent a set of objects (class extent) of one particular object type (member type). Multiple classes can be defined with the same member type, such that an object can be member of multiple classes. The member objects of a class can be restricted by a class predicate, given in the form of query expressions. The fulfillment of this predicate is a necessary precondition for that an object can be member of that class. From the separation of classes from object types follows a separate class hierarchy. The most general class of each database `DB_i` is `Objects@DB_i`. It contains all objects of that database and is defined with member type `object@DB_i` and class predicate "true". Further examples are

---

\(^2\)Performance issues are not the main concern of this paper. However, we are well aware that in addition to the known fact that late binding is a costly issue, calling functions on different databases is an even more expensive operation. It requires an object-oriented communication infrastructure for remote methods calling, like for example proposed by CORBA [COR93].
class Theatres : event;
class SwissairFlights : flight some Flights where code = “SR”

Class Theatres is defined with member type event and class SwissairFlights is defined using a predicate, such that objects of that class must be member of class Flights and with code “SR”. Predicates are always necessary but not sufficient. Hence, objects are added to classes by explicit update operations, like for example presented later in Section 4.

Similar to object types, the most general class of multiple distributed databases is called Objects@GDB, with member type object@GDB and class predicate “true”. It contains the union of all objects of the involved databases. Notice that Objects@GDB is superclass of all local classes Objects@DBi.

Given this global class hierarchy, further classes can be defined that span multiple databases, for example,

At DB3:   class Destinations : destination some Cities@DB2

This expression defines a class Destinations in DB3 as a subclass of Cities@DB2, such that all objects of Destinations are also member of Cities. Creating instances of such a global class is not straightforward. We refer to Section 4 for further discussion.

Variables are temporary handles for objects. They are used similarly to programming languages for referencing objects. Variables are defined with a name and domain type:

\[
\begin{aligned}
\text{var zurich : city;} \\
\text{var favoriteOperas : set of opera}
\end{aligned}
\]

Similar to classes, global variables can be defined using global types, and hence holding objects from multiple databases, like for example,

At DB3:   var myempls : set of employee

3.7 Database-Spanning Views

Views are the last building block of the distributed object model, and as we will see, a fundamental distribution principle. We consider views as derived classes, whose class extent and member type is derived by query expression (cf. views in relational databases). The precondition for this form of views is an object preserving semantics of query operations (see Section 4 below). This means that the objects of the view are the same objects of the base classes, and not copies of them. This makes views updatable [SLT91] because updates to objects in the view are identical to updates to the same objects in the base class. Vice versa, updates to base class objects are immediately visible in the view.

The following example defines a view ZurichFlights by selecting those flights heading for a destination named “ZRH”:

\[
\text{view ZurichFlights as select [name(to) = “ZRH”](Flights)}
\]

The power of this view mechanism becomes more clear, when we describe the query language in the next section in detail.

Views are very important in distributed object database systems, since they allow for the definition of integrating views of the distributed database. In fact, a lot of research already pointed out that the availability of views in multi-database systems is important [SST94]. Consider for example
At DB3:

\[
\begin{align*}
\text{view AirportCities as Destinations@DB1 intersect Cities@DB2;}
\text{view US_Bookings}
\text{as extend booked_US_flights = select[country(to(f)) = "US"]}
&(f: booked_flights(b))
\end{align*}
\]

The former expression defines view AirportCities as the intersection of objects from Destinations@DB1 and Cities@DB2. The later expression defines an extend-view US_Bookings, by adding a new derived function booked_US_flights to class Bookings@DB3.

4 COOL*: A Scalable Distributed Object Database Language

The distributed database system architecture determines to a large degree the object database language. We concentrate on this issue and show the differences in terms of concepts and power between languages for each architecture level 0 to IV.

Instead of presenting a different language for each level, we use again the object database language COOL*. This language can be tailored according to the underlying architecture. To our knowledge, this capability is unique to COOL*. Hence, we call it a scalable distributed object database language, meaning that COOL*(n), with n = 0, ..., IV, is the implementation for an architecture with integration level n.

4.1 COOL*(0)

Level 0 databases are not integrated at all (cf. Section 2.1). Since global query and update operations require at least some minimal form of a common schema, multi-database querying and updating is strictly limited to operations that can be performed on one single local database. No common type system is available, such that even simple query expressions spanning multiple databases could not be parsed. COOL*(0) is therefore identical to database languages for non-distributed databases, like for example COOL or ODMG-93.

Database cooperation is completely up to the application programmer, who might use distributed transaction processing. Consider the following transaction. A query statement finds the name of a city in database DB2 and stores it in a programming language variable c of type string. Based on that, another query expression in database DB1 searches for Destinations with the same name.

\[
\begin{align*}
\text{var c: string;}
\text{BEGIN_OF_TRANSACTION;}
\text{c := name(pick(Cities@DB2));}
\text{select[name(d) = c](d: Destinations@DB1);}
\text{BEGIN_OF_TRANSACTION}
\end{align*}
\]

This simple kind of transaction based interoperability is using the compatibility of basic data types. Consequently, we already require at this level that basic data types are compatible among databases (cf. Section 3.1) or that their exist equivalence preserving transformation operations. Otherwise, even such simple transactions would not be possible. Beyond that exchange of data values, no further global operations are possible.
4.2 COOL*(I)

In level I federated databases, schema composition is available forming the basis for further integration (cf. Section 2.2). Hence, COOL*(I) adds a single new import statement to the language, importing the schema of another database and putting it side-by-side with the local schema. For example, database DB3 can be made a level I federated database as follows:

At DB3: \texttt{import DB1}, \texttt{DB2}

A global type system with types \texttt{object@GDB} and \texttt{bottom@GDB} is created at DB3 as described in Section 3.4 such that from now on, type expressions are possible that include multiple databases. Furthermore, global common superclass \texttt{Objects@GDB} is created. No further connections are established between the schemas, neither additional classes, nor global functions or views.

Schema composition merges the name spaces. Holding the names of all schema elements in the distribution dictionary, they are known to the importing database. As a consequence, the above COOL*(I) transaction can now be rewritten at DB3 as a single COOL*(I) query:

At DB3: \texttt{select[name(d) = name(pick(Cities@DB2))]}(d: Destinations@DB1)

This query is correct at level I and can be parsed by database DB3. Another example of a correct inter-database query is the following intersection of objects from different databases:

At DB3: \texttt{Destinations@DB1 intersection Cities@DB2}

Beyond that, COOL*(I) capabilities are very limited. For example, there is still no definition of inter-database functions and therefore there are no same-functions to integrate objects (cf. Section 3.2). Objects from multiple databases can only be compared based on primitive data types. Though inter-database queries like the above intersection are correct, the resulting set is empty by definition, because so far no two objects from databases DB1 and DB2 can be the same.

Global update operations are only possible, if they can uniquely be mapped to one database. Notice, that this imposes major restrictions. For example, while creating new objects in DB3, the database in which the object will be stored must explicitly be specified. The following correct operation creates new object \texttt{o1} of type \texttt{city} in database DB1.

At DB3: \texttt{create[city@DB2](o1)}

Although the update is expressed at DB3, it can completely be performed in DB2. The same operation, but with the previously defined global type \texttt{employee} is not correct in COOL*(I), since it is a distributed type and requires a notion of distributed objects. Consider another valid update operation:

At DB3: \texttt{add[o1](Events@DB2)}

It adds city object \texttt{o1} into class \texttt{Events}. This would not have been allowed, if the target class stemmed from a different database than DB2, since objects of one database cannot (yet) be added to other databases.

\textsuperscript{3}Notice that we consider a symmetric situation, where each database can import from any other database. Hence, there is no selected global database and DB1 or DB2 could import the other’s schemas as well.
4.3 COOL*(II)

Level II federated databases can virtually be integrated. Hence, the concept to be added for COOL*(II) are global views that span multiple local databases (cf. Section 3.7). As already mentioned, views are considered virtual classes whose class extend and member type is derived by a query. The definition of these global views are stored in the distribution dictionary. Reconsider the above mentioned inter-database query, now used for the definition of a view

At DB3: \[ \text{view } V \text{ as } \text{Destinations}@DB1 \text{ intersection Cities}@DB2 \]

Inter-database views can be employed to further integrate and change the schemas. Most important is that the availability of views at level II allows already for the integration of objects, that is, to define \textit{same}-functions. Consider for example the only \textit{same}-function illustrated in Figure 3. It can be defined using a COOL*(II) view as

At DB3: \[ \text{view } \text{Destinations}@DB1' \]
\[ \text{as extend}[\textit{same} := \text{select}[\text{name}(c) = \text{name}(d)](c: \text{Cities}@DB2)] \]
\[ (d: \text{Destinations}@DB1) \]

The effect of this view is dramatic: From now on, two objects from databases DB1 and DB2 are treated by COOL* operations as the same, if they have the same name. As an immediate consequence, the set of objects in the above view \( V \) is not anymore empty, if there are objects in \text{Destinations}@DB1 and \text{Cities}@DB2 having equal name.

Though these \textit{same}-functions must be expressed using views and are therefore derived functions only, this is a powerful additional possibility. By definition, all query operations respect the special semantics of \textit{same}-functions and treat two objects from different databases that are integrated using a \textit{same}-function as the same object. The availability of \textit{same}-functions adds the full functionality to distributed query operations.

With respect to update operations, COOL*(II) does not extend the functionality of the previous level, hence the same restrictions apply. The capability of views, and in particular of derived \textit{same}-functions, is not powerful enough to achieve full update functionality. To summarise, queries now have their full global functionality at this level. In contrast, updates are still restricted.

4.4 COOL*(III)

At level III, integration of databases is not anymore restricted to views. Hence, COOL*(III) adds the possibility of performing any kind of global schema extensions, for example to create stored inter-database functions or classes (cf. Sections 3.3 or 3.6). As an example, a class \textit{Employees} can be defined at DB3 as a subclass of \textit{Persons} from DB1 and with global member type \textit{employee}, that was defined earlier using functions from DB1, DB2, and DB3 (see Section 3.5)

At DB3: \[ \text{class } \text{Employees : employee some } \text{Persons}@DB1 \]

Globally stored \textit{same}-functions are possible. Consider as an example the above derived \textit{same}-function, defined using a view. With COOL*(III) this \textit{same}-function can be defined as a stored inter-database function, which was not possible at the previous level:
At DB3: \(\text{function } \text{same} : \text{destination} @ DB1 \rightarrow \text{city} @ DB2\)

To store the values of such database-spanning functions, types, and classes, the distribution dictionary gets the additional possibility of storing not only schema definitions, but primary objects of any database too.

Stored \textit{same}-functions dramatically extend the power of the COOL*(III) with respect to update operations, as compared to the previous level. For this purpose, we do not invent new operations but extend the semantics of the existing centralised update for the management of distributed objects, using the revised definition of distributed object identity (Definition 1) based on \textit{same}-functions. Each of these distributed generic update operations can be seen as a distributed transaction for two reasons: first, they are performed atomically, that is, as a whole or not at all, and second, their impact must be synchronised with local database transactions.

We briefly sketch the distributed generic COOL* update operations, concentrating on how they work in a distributed object management system and why they have not been possible at the previous level. The complete formal definition of the distributed update operations is given in Appendix A.

Consider first global variable assignment that spans multiple databases

At DB3:
\[
\begin{align*}
\text{var } c : & \text{city} @ DB2; \\
& c := \text{pick}(\text{select}[*]*[(\text{Destinations} @ DB1)])
\end{align*}
\]

Variable \(c\) is of database DB2, whereas the assigned object \(o\) is of DB1. If the assigned value were a primitive value, this assignment could directly be performed because primitive data types are compatible by definition. However, if the right side is an object, it cannot be performed within one database. Instead, the distributed assignment operation must assigns the \textit{same}-object of \(o\) in database DB2 to the variable (cf. Figure 5(a)).

In order for the assignment \(c := o\) to be correct, the type of \(o\) must be a subtype of the type of variable \(c\). More formally, \(c : t\) and \(o : t'\) and \(t' \leq t\) (which must be satisfied independent of whether a distributed database is considered or not). According to type inference rule of Section 3.2, if \(c\) and \(o\) stem from different databases, this precondition can only be satisfied if there exists a \textit{same}-object of \(o\) in database 1. Consequently, there is always a \textit{same}-object of \(o\) in DB1.

Similar situations occur for partial assignment of a new value to a function. In the distributed travel database system, function \textit{from} is for example defined in the database DB1, object \(f\) in DB1 and expression \(c\) in DB2.
As sketched in Figure 5(b), this distributed operation is always performed in database DB1, where function from is defined. If c and/or f stem from other databases, their same-objects in database DB1 are used for assignment.

Again, there are typing preconditions, namely from : $t_1 \rightarrow t_2$ and $f : t' \leq t_1 \leq \text{object}_o$ and $c : t$ and $t \leq t_2$, from which follow that such same-objects always exist in database DB1. If c is a set of objects, taking the same-object for each member object must be considered.

Object evolution determines updates that dynamically change the type of an object, by adding new types to the object, gain[t](v), or removing existing types from objects, lose[t](v). In a distributed database, a city object in a variable of database DB2 may gain the type destination of DB1,

At DB3: \[ \text{var } c : \text{city@DB2}; \]
\[ \text{gain[destination@DB1]}(c) \]

The semantics of distributed gain can best be explained by making use of the fact that each object type is given as a set of functions, $t = [f_1, \ldots, f_n]$. Hence, for each function $f$ in type $t$, gain[$f$](v) can be performed in the database where $f$ was defined. As sketched in Figure 6(a), if $f$ stems from a different database than $v$, the gain operation is performed on $v$'s same-object in that database.

Figure 6: (a) Type evolution and (b) class evolution in distributed ODBMS

There is no evidence that these same-objects must already exist. That is, the distributed gain-operation may require to generate new proxy objects and to generate new same-functions in between. This is exactly what has not been possible at the previous level, using derived same-functions only.

The lose operation may involve multiple database systems too. For example, to reverse the above operation

At DB3: \[ \text{lose[destination@DB1]}(c) \]

In a distributed database, for each function $f$ of type $t$, a local lose-operation is performed in the database where $f$ was defined. If this database is different from where $v$ is
defined, the operation is performed on the *same*-object. In contrast to the *gain* operation, the type preconditions for *lose*, \( o :: t' \) and \( t' \leq t \leq \text{object}_G \), guarantee that such a *same*-object exists.

Class extents are modified independent of changing the type by update operations to add and remove objects from the class extent. Adding an object to a class \((\text{add}[o](c))\) automatically adds this object to all super-classes. Removing an object from a class \((\text{remove}[o](c))\) means at the same time, removing this object from all subclasses.

In a distributed object database system, one must allow for adding an object to any class in any local database. Consider for example the following expression that adds an object of database DB2 to a class of DB1:

```
At DB3:
var m: musical@DB2;
add[m](Destinations@DB2)
```

Since the object \( m \) stems from a different database than the target class, the *same*-object of \( m \) in database 1 is added to class *Destinations* (cf. Figure 6(b)). It may be necessary that this *same*-object must be created first because it does not already exist.

The same situations must be distinguished for the *remove* operation. If class *Destinations* and object \( m \) stem from different databases, the *same*-object of \( m \) in the target database is removed from class. The global typing preconditions, \( m :: t \) and \( t \leq \text{object}_G \), guarantee that this *same*-object already exist.

There are updates that create \((\text{create}[v](v))\) and delete objects \((\text{delete}(v))\) of an object type \( t \). Creating an object in a distributed object management system always involves multiple steps because type \( t \) of which it is created may be a subtype of multiple databases. Hence, an object is first created in the database where variable \( v \) is defined. Then for each database of which \( t \) is a subtype, a *same*-object is created. Finally, a distributed *gain*-operation is performed, which makes all *same*-objects an instance of type \( t \).

The expectation for a distributed delete operation is that all proxy-objects of a global object are completely removed from the distributed database. Hence, *delete* not only deletes the object it was called on, but also all *same*-objects of that object. This semantics of distributed delete is inverse to the above presented distributed create operation.

To summarise, global update operations have their full functionality too at integration level III. The same operations that are available for a local database can be used on the global level. Particularly interesting are distributed update operations, that have been restricted at the previous level. Objects of one database can be added to another database or may get a type of another database, and global objects that are partitioned among multiple local databases can be created. The prerequisite for this power is, that if necessary, update operations can define two objects as being the same using stored *same*-functions, which has not been possible at the previous level.

### 4.5 COOL*(IV)

At level IV, integration of database systems into one global system is complete, or in other words, there is no logically distributed database system. Logically, there is one single domain of objects, such that no *same*-functions are necessary. Even if objects are physically distributed over multiple disks, this is an issue of physical database design, and *same*-functions are not the concept to model these aspects.
Hence, COOL*(IV) is identical with a language for centralised database systems. The operators, their interface, and their power is the same as those of COOL. However, the implementation is different, since it does not have to consider autonomy of local databases.

Of course, query and update operations have their full functionality, since there is only one database management system, processing the operations.

5 Related Work

The work presented in this paper extends and formalises previous work on using views defined by algebraic query expressions to integrate not only schemas but also objects from multiple databases [SST94]. It is emphasised on which language power can be achieved by using inter-database views as the main mechanisms for cooperation. It is discussed, how meta-databases are integrated to achieve schema integration, using the same view mechanisms on the meta database. In contrast, we focus in this paper on principles for languages and consider in particular update operations too.

Other projects, where views have been of interest for cooperation of loosely coupled databases are Superviews [Mot87], MultiBase [LR82], and VODAK [NS88]. It is remarkable that these systems are retrieval-only, i.e., they do not allow for updates.

Several proposals for comprehensive multi-database languages can be found in the literature. [Kli91] presents language features for the interoperability of databases with semantic discrepancies. Pegasus [AAD+93] even offers a concept called image-function for merging objects from different databases into a global object. Similarly, in O*SQL [Lit92] a merge-operation is proposed. For Pegasus, we did not see precise definitions and maintenance of distributed object identity and O*SQL lacks of formal definitions for the operations, which makes particularly updates difficult to trace.

We refer to [TS94], where several multi-database languages were classified according to their cooperation mechanisms. This paper also shows, how SQL databases achieve some preliminary form of distribution, like for example Oracle SQL*Net [SQL89] and INGRES/Star [Ing91].

6 Conclusion and Future Work

The contribution of this paper is a formal discussion of principles for distributed object database languages. Since these principles are independent of any concrete database language, they serve as guideline for extending a non-distributed object database language, like for example an object-algebra, object-extensions to SQL, or ODMG-93, towards the management of distributed objects. The principles of distributed object database languages are:

- Primitive data types, like for instance integer, real, or string, must be compatible among databases. This is the precondition for that distributed object database languages can compare values from different databases.

- There must be a notion of distributed object identity, allowing for integrating proxy objects from different databases that represent the same real world entity object.
Such integrated objects must be treated by the distributed object database language as one single object, allocated over multiple sites.

- Relationships among objects from different database must be definable in order to establish inter-database integrity constraints, and in particular, to link proxy objects that represent the same real-world entity. These global integrity constraints must be respected by the distributed object database language.

- A global type lattice must be available for static and dynamic type checking of distributed object database operations. Implicit detection of subtypes based on conformity of type interfaces achieves additional flexibility when independently defined schemas are merged.

- Overriding and data encapsulation provide a high level of transparency to distributed object database languages. Functions defined in one database may be overridden in another database and encapsulation may hide the location of method implementation.

- Views are necessary to stepwise abstract from separated distributed databases into a global schema, where distribution is more transparent. This schema must be extensible with database-spanning classes and variables to enrich merged databases.

It was formally shown that the power of distributed object database languages depends on the underlying architecture, that is, on the degree of preserved database autonomy. To illustrate the implementation of the presented principles, COOL* was introduced, a scalable distributed object database language, that can be tailored for a particular architecture. More precisely, for five given architectures with increasing degree of global control and decreasing degree of local autonomy, the corresponding language COOL*(0) to COOL*(IV) were sketched.

COOL*(0) is the basic object database language for non-distributed database systems. Because it does not provide any distribution mechanisms, interoperability must be hand-coded using distributed transactions. COOL*(I) adds schema composition, a mechanism to create a minimal global schema and merge name spaces. Global queries can be formulated. COOL*(II) adds a global view mechanism to define database spanning views and hence virtually integrating local databases. Global queries can be formulated that respect integrated objects. COOL*(III) adds the possibility for real integration of local schemas, by creating any global schema element. Global updates can be formulated that respect integrated objects. COOL*(IV) is the basic object database language for non-distributed database systems. The distribution is a matter of physical database design and does not affect the language interface.

**Future Work**

The design of COOL* is part of an overall ongoing project that aims to develop a middleware system for distributed object management. Our vision is, that this middleware will export comprehensive database functionality to objects stored in external repositories, that may not provide such functionality by themselves.

A prototype Concert of this distributed object middleware system is currently under development. Whereas major parts of the basic open abstract-object storage system
[BRS96] have been implemented, we are currently working on extending its programming interface with the algebraic COOL* distributed object language. We aim to support the full flexibility of repository autonomy, i.e., depending on the degree of autonomy one chooses, the middleware provides a more or less powerful implementation of the COOL* language.

We have on-going work in the areas of incorporating transaction management and external data structures into the middleware. We will treat the question of how to make the middleware language open for advanced query capabilities provided by the repositories, for example, how to extend it towards spatial or multimedia querying. Hence, we are applying the ideas described here in the context of multimedia and geospatial projects.

Acknowledgements

We are indebted to Hans-Jörg Schek and Marc Scholl who contributed in many discussions to the design of COOL*. We thank Michael Rys and Uwe Röhm for their thoughtful comments on this paper.

References


A Definition of Distributed Update Operations

A procedural semantics is given to formalize the definition of all primitive update operations for distributed object databases. The approach is not to introduce a new database language with new operations, but to take a well defined language for non-distributed object databases, in our case COOL, and to extend it for the management of distributed objects.

In the sequel, it is shown how each of the distributed operations is implemented by a procedure using the non-distributed operations. This procedure must be understood as a distributed transaction. For each operation, typing preconditions are given. $t' \subseteq t$ is true, if $t'$ is a subtype of $t$ or $t'$ and $t$ are the same type under conformity.

Operations performed locally on database $DB_i$ are marked with index $i$, like for example $\text{gain}_i[f](o)$. Distributed operations are marked using a $\ast$, like e.g. $\text{gain}_i^\ast[f](o)$. Notice that this is to clarify only, and that the language needs neither of these marks.

We use a function $\text{where}$ defined as

\[ \text{function where} : \text{object} \rightarrow \text{integer} \]

as $\text{where}(o) = i \Leftrightarrow \text{object}_i(o)$

that returns for a given object $o$ the number $i$ of the database $DB_i$, where object $o$ has been created.

Variable Assignment

Global operation:

\[ e := t \]

Typing preconditions:

\[ e : t, e : t', t' \subseteq t \]

Mapping to local operations:

\[ \text{FOREACH } j = 0..n : v_j = v \land v_j = \text{same}_j(v) \text{ DO} \]

\[ \text{IF } t \subseteq \text{object}_i \text{ THEN} \]

\[ i := \text{where}(e); \]

\[ \text{IF } j = i \text{ THEN } e := v_j \]

\[ \text{ELSE } v_j := \text{same}_j(v) \text{ END} \]

\[ \text{ELSE IF } t \subseteq \text{object}_i \text{ THEN} \]

\[ v_j := \{ \}; \]

\[ \text{var } e_j : \text{object}_i; \]

\[ \text{FOREACH } e' \in e \text{ DO} \]

\[ e' := \text{set} ; \]

\[ e' := e' \cup \{ v_j \} \]

\[ \text{END} \]

\[ \text{ELSE } v_j := e \text{ END} \]

\[ \text{END.} \]

Partial Assignment

Global operation:

\[ \text{set}^* [f := e](o) \]

Typing preconditions:

\[ f : t_1 \rightarrow t_2, o : t', t' \subseteq t_1 \subseteq \text{object}_i, e : t, t \subseteq t_2 \]

Mapping to local operations:

\[ \text{FOREACH } j = 0..n : f_j = f \land f_j = \text{same}_j(f) \text{ DO} \]

\[ \text{IF } t \subseteq \text{object}_i \text{ THEN} \]

\[ i := \text{where}(e); \]

\[ \text{IF } j = i \text{ THEN } \text{set} [f_j := e](o) \]

\[ \text{ELSE } \text{set} [f_j := \text{same}_j(e)](o) \text{ END} \]

\[ \text{ELSE IF } t \subseteq \text{object}_i \text{ THEN} \]

\[ \text{set} [f_j := \{ \}](o); \]

\[ \text{var } e_j : \text{object}_i; \]

\[ \text{FOREACH } e' \in e \text{ DO} \]

\[ e' := \text{set} ; \]

\[ e' := e' \cup \{ f_j \} \]

\[ \text{END} \]

\[ \text{ELSE } v_j := e \text{ END} \]

\[ \text{END.} \]

Gain Object Type

Global operation:

\[ \text{gain}^* [t](o) \]

Typing preconditions:

\[ o : t', t = [f_1, \ldots, f_m], t' \subseteq \text{object}_i \]

Mapping to local operations:

\[ \text{FOREACH } f = [f_1, \ldots, f_m] \text{ DO} \]

\[ \text{FOREACH } j = 0..n : f_j = f \land f_j = \text{same}_j(f) \text{ DO} \]

\[ i := \text{where}(e); \]

\[ \text{IF } j = i \text{ THEN} \]

\[ \text{gain}^*_j[f_j](o) \]

\[ \text{ELSE IF } \text{same}_i(o) = \omega \text{ THEN} \]

\[ \text{set}^* [\text{same}_i := \text{new}_i][\text{object}_i](o) \text{ END}; \]

\[ \text{gain}^*_j[f_j][\text{same}_i(o)] \]

\[ \text{END} \]

\[ \text{END.} \]
Lose Object Type

Global operation:
\texttt{lose}^[t][o]

Typing preconditions:
\(a: t', t = \{f_1, \ldots, f_m\}, t' \leq t \leq \text{object}_o\)

Mapping to local operations:
\begin{verbatim}
FOREACH \(f = f_1, \ldots, f_m\) DO
  \(i := \ast \text{where}(o);\)
  \text{IF } j = i \text{ THEN } \text{lose}_{[f_j]}(o)\)
  \text{ELSE } \text{lose}_{[f_j]}(\text{same}_{i, j}(o)) \text{ END}
\end{verbatim}

END.

END.

Object Deletion

Global operation:
\texttt{delete}^[o]

Typing preconditions:
\(a: t, t \leq \text{object}_r\)

Mapping to local operations:
\begin{verbatim}
i := \ast \text{where}(o);\)
\text{IF } j = i \text{ THEN } \text{delete}_{[f_j]}(\text{same}_{i, j}(o)) \text{ END;}
\text{delete}(o).
\end{verbatim}

Add to Class Extent

Global operation:
\texttt{add}^[o][c]

Typing preconditions:
\(a: t, t \leq \text{object}_o\)

Mapping to local operations:
\begin{verbatim}
i := \ast \text{where}(o);\)
\text{FOREACH } j = 0..n : c_j = c \lor c_j = \text{same}_{j}(c) \text{ DO}
  \text{IF } j = i \text{ THEN}
    \text{add}_{[c_j]}(c_j)
  \text{ELSE}
    \text{IF } \text{same}_{i, j}(o) = \omega \text{ THEN}
      \text{set}^[\text{same}_{i, j} := \text{new}^[\text{object}_r][j](o)](o) \text{ END;}
    \text{add}_{[\text{same}_{i, j}(o)][c_j]}
\text{END}
\text{END.}
\end{verbatim}

Remove from Class Extent

Global operation:
\texttt{remove}^[o][c]

Typing preconditions:
\(a: t, t \leq \text{object}_o\)

Mapping to local operations:
\begin{verbatim}
i := \ast \text{where}(o);\)
\text{FOREACH } j = 0..n : c_j = c \lor c_j = \text{same}_{j}(c) \text{ DO}
  \text{IF } j = i \text{ THEN}
    \text{remove}_{[c_j]}(c_j)
  \text{ELSE}
    \text{IF } \text{same}_{i, j}(o) \neq \omega \text{ THEN}
      \text{remove}_{[\text{same}_{i, j}(o)][c_j]} \text{ END}
\text{END}
\text{END.}
\end{verbatim}

Object Creation

Global operation:
\texttt{create}^[t][v]

Typing preconditions:
\(v: t', t \leq t' \leq \text{object}_o\)

Mapping to local operations:
\(\text{new}^{[\text{object}_r]}() \rightarrow \text{new}().\)