UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK

Postfach 6980, D-76128 Karlsruhe

Multi-Object Cooperation in Distributed Object Bases

Dietmar Kottmann, Peter C. Lockemann, Hans-Dirk Walter

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
Fakultät für Informatik
D-76128 Karlsruhe, F. R. G.

Interner Bericht Nr. 16/95

Abstract

It is an emerging trend to build large information systems in a component-based fashion where the components follow the concept of object. Applications are constructed by organizing pre-built objects such that they cooperate with each other to perform some task. However, considerable programming effort is required to express multi-object constraints in terms of the traditional message-passing mechanism. This observation lead many authors to suggest communication abstractions in object models. One promising approach is to separate multi-object constraints from the objects and collect them into a separate construct. We call this construct an *alliance*.

Unlike other approaches we allow alliances to involve large sets of long-lived objects which may dynamically vary during the — also potentially long — life-time of the alliance. Alliances are not only visible at the specification level but are also computational entities which enforce multi-object constraints at run-time. They do so in an unreliable world, i.e., we do not assume that objects will always meet their obligations in a cooperation.

Since objects may often be distributed across a network, we demonstrate that alliances are an ideal place to deal with aspects of distribution in an application-specific manner. We illustrate our thesis by one of the key questions of distributed object management: where shall objects be located and when shall they migrate to which node? We show that alliances allow for customized distribution policies which are neither "hardwired" into the objects nor necessitate a centralized distribution control.

Keywords: cooperation, distribution, object-oriented databases, dynamic constraints

This work was partly supported by the German Research Council (Deutsche Forschungsgemeinschaft DFG) under grant SFB346.

1 Introduction

It is an emerging trend to build large information systems[1] in a component-based fashion where the components follow the concept of object. Modern standardization efforts as, e.g., OMG's Corba [29], echo this trend. Applications are constructed by organizing pre-built objects such that they cooperate with each other to perform a given task. An object is usually described by an identifier and an explicit interface which is restricted to a set of unrelated messages through which other objects can interact. Its state and implementation are hidden.

The cooperation between objects must follow task-specific rules that go beyond the rules which govern the behavior of an individual object. Consider, e.g, a travel scenario where objects implementing one or more hotels, airlines, and the customer interact to satisfy a customer's reservation needs. This may require rescheduling of flight reservations because of the unavailability of hotels, calling up the hotels in order of preference, or have fully booked hotels find a substitute hotel as a special customer service, and not the least, of course, inquiries with the customer him/herself to determine whether to continue at all or in the planned fashion. All this should not just proceed in an arbitrary fashion but obey certain constraints. Such a set of constraints is often referred to as a protocol.

It has been recognized for several years that considerable programming effort is required to express multi-object constraints such as temporal ordering of messages in terms of the traditional message-passing mechanism. In essence, expressing the constraints by explicit message passing "hardwires" the constraints into the object implementation and, for that matter, spreads them across multiple implementations. If we consider that an object may participate in a number of tasks which differ in their constraints, object implementation may become overloaded, difficult to understand, and, hence, prone to errors that are extremely difficult to dissect and correct. It further obstructs reusability of objects — a strength often claimed for object-oriented models. A programmer also must anticipate all possible "misbehaviors" of cooperation partners. Otherwise, the object state may be left inconsistent. For instance, in a travel information system hotel objects must provide code to deal with clients unwilling to pay rented rooms. In complex environments such misbehavior may not always

[1]We define an information system by the equation *information system = applications + database(s)* as in [8].

be predictable at the time the objects are implemented.

In order to overcome these deficiencies a promising approach seems to separate the constraints from the objects into communication abstractions as has been proposed, e.g., in [2, 3, 13, 21, 22, 33]. A separate construct defines a set of communication participants, each playing a certain role, and a set of constraints regulating the inter-object communication. We claim that all these approaches are too limited to deal with applications that are part of a large information system, e.g., a world-wide travel information system. Consequently, we introduce an extended construct which we call an *alliance* and which emphasizes the following aspects:

Support of large and evolving sets of participants The tasks which objects collectively perform may be complex. Just consider the travel reservation above. Tasks may be long (or in some cases infinitely) lasting and may have numerous participants some of which may not be known at the beginning of the task.

Support of long-lived tasks Both tasks and objects may be long-lived. Further, tasks may outlast objects, and objects may outlast tasks. Consequently, not only should objects be persistent but also alliances. Further, persistence of alliances should be treated independently of that of objects.

Integration with a communication subsystem Objects may be distributed across a network. Consequently, a considerable part of the communication may take place across communication lines, with all the associated problems. Since a wealth of supportive high-level communication protocols [15] and distributed middleware [29] is available, we propose to view alliances akin to a protocolled communication medium in order to develop a seamless solution along the lines of a multi-layered architecture [15].

Enforcement of protocols In an object world one observes a large degree of concurrency both across objects and for the service calls on an individual object. For the latter — local concurrency — we assume that decisions are strictly the responsibility of the object. For the former we do not require that local concurrency policies of objects follow a global scheme as, e.g., in distributed transaction processing environments [23].

Rather we propose that alliances will deal with concurrency between objects in a task-specific fashion. Further, alliances will have to enforce constraints on temporal ordering of messages and constraints on message parameters. They cannot assume that objects will always meet their obligations in a cooperation, and they will have to initiate compensating actions in case of constraint violations. Also, since it is impossible to treat all misbehavior of the underlying hardware generically [24], application-specific counter-measures become necessary. Therefore, alliances also need some kind of "active behavior", a local state to store some context information, and their own execution model.

Distribution policies Optimization of quality parameters, such as performance or resilience to network failures or security, are governed by distribution policies over base mechanisms as object migration or replication. We claim that alliances are the ideal place to deal with those questions because distribution should remain entirely transparent to the objects and, thus, should be taken largely outside the objects.

Integration with an object model We presume as usual that the majority of information processing takes place locally which implies that objects remain the carrier of all major actions. They will initiate alliances and instigate all actions within them. Consequently, at least some objects need some knowledge about alliances.

On the other side in a large information system a large number of objects will take part in many different tasks at unknown times and, hence, should be oblivious to their participation in alliances. In other words, these objects should still be able to call upon the services of other objects in the classical manner, and observe the existence of multi-object constraints solely by whether their service calls are successful or not.

By necessity, objects that run at different nodes and, hence, in separate processes, have a large degree of autonomy over their decisions. It seems natural to introduce *concurrent objects* or *actors* [1] (sometimes also referred to as *active objects* [26] or — nowadays — as *agents* [6]) to capture this autonomy [5]. In this way we follow the approach of [2] and assume *autonomous objects* as participating objects[2]. Object autonomy implies that we do not make any assumptions about the computational model of objects.

[2]In fact the four notions — *concurrent object*, *actor*, *active object*, and *agent* — have not an identical meaning in the literature, for some there is no commonly agreed meaning, or the notion may be misleading. We do not wish to enter into a discussion on their relative merits and instead use a more neutral term.

The remainder of the paper is organized as follows. In Section 2 we compare our work to related research. In Section 3 we introduce the alliance model. Section 4 discusses how alliances can support application-specific distribution policies. Section 5 deals with the issue how to integrate alliances into a given object model. In Section 6 we introduce our prototype architecture and discuss first practical experiences. Section 7 concludes this paper and suggests topics for further work.

2 Related Work

It has long been recognized that communication abstraction is necessary in object-oriented models. One class of approaches extends interface descriptions of objects by synchronization constraints (e.g., [18, 19, 26, 31]) or by a declarative description of object behavior (e.g., by using finite state machines as in [33]). In some cases the separation of interface and implementation was completely abandoned (as e.g., in [34]). All these approaches limit themselves to object-specific synchronization — we called objects with this capability autonomous objects — but continue to treat objects as islands, and thus do not touch on the problems mentioned in Section 1.

Active objects in active object-oriented database systems (OODBS), as, e.g., in [7, 9, 10, 11], are able to detect events and to execute — also asynchronously — some predefined code as a reaction. But they are not able to limit method invocations. One can interpret the raising and detection of an event as a communication between raising object and detecting objects. Following this interpretation, an object that raises an event "broadcasts" some information to all objects that are interested in that event — which is specified by an appropriate trigger as part of the object implementation. Consequently, besides the directed method invocation active OODBS offer the anonymous broadcast as a second communication paradigm. Unfortunately, this form of communication is largely unregulated and indiscriminate, and any control over the communication is by purely local condition checking. This is a far way from our target to allow for arbitrary but controlled multi-party communication patterns.

Today, transactions are the most common means to guarantee multi-object consistency [12]. Transaction concepts define consistency more or less independent from application semantics. In most cases correctness is based on serializability or some extension of it. Therefore, all objects must obey a globally defined synchronization scheme [23]. Consequently, transaction concepts limit object autonomy and impose a fixed protocol that cannot be adapted to task-specific constraints on temporal orderings of messages in the context of an activity. These constraints remain hidden in the implementation of the participating objects. There is a bit more flexibility in script-based approaches (e.g., [27, 32]) that define a "work-flow", but they require a rigorous and complete a-priori definition of the ordering of transactions and method invocations, thus denying all evolution. However, transactions can be expected to play an important role in an implementation of alliances.

Interoperable transactions [25] provide a language based on temporal logic to specify the temporal ordering of messages between a group of cooperating objects. The participants in an interoperable transaction are determined at the beginning of a cooperation and cannot change later on. The approach is exclusively intended for specification and verification of cooperation protocols. Nothing is said about an implementation of a cooperative application specified in the proposed language. Consequently, integration with a communication subsystem in a distributed environment and compensation of protocol violations are not considered.

Similar arguments hold for the concept of *connectors* [3], a CSP-based formal description language for software architectures, since this concept is also restricted to the specification level. Connectors specify interactions between a fixed number of software modules. Consequently, enforcement of protocols at runtime or distribution aspects are not part of this research.

Closest to the intention of our approach are *contracts* [13, 14], *synchronizers* [2], and *adaptors* [33]. Each of them collects some aspects of an intended cooperation into a separate construct which has also a run-time representation. A contract defines a set of communicating participants — which must be completely known at the time of the contract's instantiation — and their contractual obligations. Contracts are not intended to define multi-object constraints but utilize their contexts to describe the behavior of a participating, i.e. their methods are required to conform to the contract.

A synchronizer simply limits the invocations accepted by a group of objects. Adaptors allow for the behavioral composition of two objects, which are functionally but not necessarily type compatible. In contrast to synchronizers adaptors are not restricted to the limitation of method invocations but have some limited control over messages as well. For instance, they can map messages between sender and receiver or they can synthesize a set of messages originating from a sender object into a single one which is actually delivered. Therefore, adaptors are equipped with their own memory. Adaptors are restricted to two participants.

Synchronizers and adaptors can be integrated with an object model without touching the object paradigm. Both models support autonomous objects — which is in contrast to contracts. All three models are restricted to a fixed number of participants which cannot evolve during a cooperation. None of the models deals with persistence or distribution. [2] mentions distribution but considers it strictly an implementation issue to be solved, e.g., by RPC-style calls.

3 Alliances

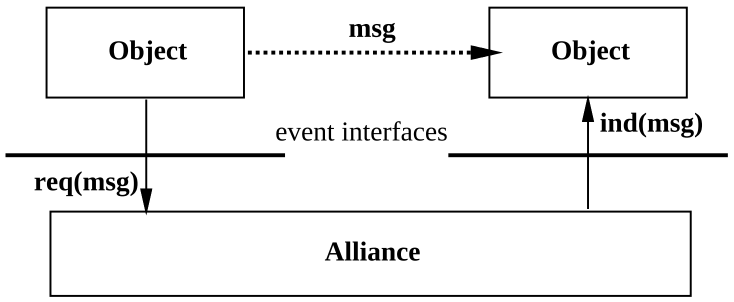

Figure 1: Alliances as communication media between objects

In extension of the terminology of the ISO/OSI reference model [15] one may view an alliance as an "intelligent" communication channel between two or more objects, which must be established between them before they can communicate. Alliances exceed this metaphor by allowing multi-object cooperation where all objects may have the same rights (in contrast to client-server models), may be long-lived, and support a wealth of semantically rich messages.

Figure 1 shows how this metaphor can be carried over into an object world: a one-way message passing (*msg*) between two objects is mapped onto two events — message request (*req(msg)*) and message indication (*ind(msg)*). The sender object raises the first with the alliance. The alliances raises the second with the receiver object.

We introduce the details of the alliance model by way of a running example: the cooperation of a client and a hotel in the course of an accommodation. Figure 2 shows an abstract view of a simple cooperation protocol for this example in terms of a state-transition diagram. The labels of the transitions denote the messages which invoke transition. The general structure of a transition label is ⟨sender⟩: ⟨receiver⟩.⟨msg⟩[⟨parameter⟩]. 'c' stands for client, 'h' for hotel, "CI" for checked in. The diagram expresses what a user might consider a correct ordering of messages in the course of a hotel accommodation from reservation to checkout.

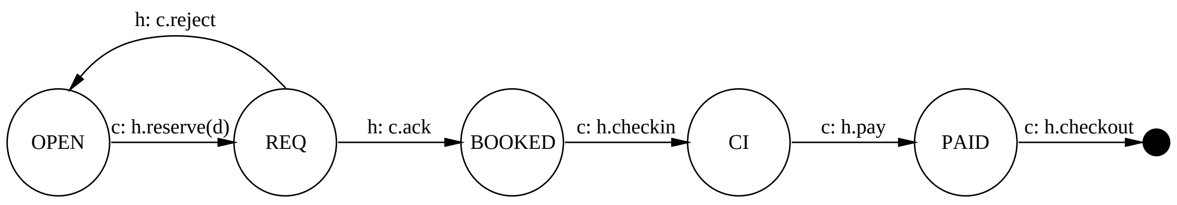

Figure 2: Dynamic model of a hotel accommodation

3.1 Alliance Types

Alliances incorporate the description of objects that can play a role in them, a set of states, the initial participants and the initial state, and a set of communication rules. Alliances with similar properties, i.e., roles, states, and communication rules, are classified into *alliance types* which can dynamically be instantiated at run-time. An example of a complete alliance type definition is given in Figures 3, 4, and 5.

An alliance type has a unique type name. The **roles**-clause determines which messages an object can receive (←) and send (→) in its role as a specific participant. For instance, an object that takes the part of a client in a hotel reservation can send the messages reserve (with message parameter of type Date), checkin, pay, and checkout and can receive the messages ack (with message parameter of type Bool) and remind. Roles can either be *single-valued* (denoted as [ ]) or *set-valued* (denoted as { }). At most one object may be bound to a single-valued role, an arbitrary number of objects may be bound to a set-valued role. Examples for single-valued roles are given in Figure 3. An example for a set-valued role — the role servers — is given in Figure 6. We will discuss set-valued roles in more detail in Section 3.4.

The **states** clause contains typed variables which defines the set of possible states. Depending on the complexity of a cooperation an alliance may have a large (possibly infinite) number of states (see, e.g., Figure 6).

```
alliance HotelAccommodation is
roles
    client: ⟵ [ack(bool); remind]; ⟶ [reserve(Date); checkin; pay; checkout];
    selHotel: ⟵ [reserve(Date); checkin; pay, checkout]; ⟶ [reply_reserve(Bool)];
    clock: ⟵ [def(int); undef]; ⟶ [alarm];
states
    progress: enum(OPEN, REQ, BOOKED, CI, PAID);
    timer: Timer;
birth (c: client, h: selHotel) is
    var h: selHotel;
    begin
        assoc(client, c); assoc(selHotel, h); assoc(clock, timer.create);
        progress:= OPEN; persistent;
    end;
rules ...
end alliance HotelAccommodation;
```

Figure 3: Specification of alliance type HotelAccommodation

A special parameterized **birth** operation allows to define the initial state and the initial set of participants of an alliance. It is automatically executed at instantiation time. Figure 3 shows that both the client- and selHotel-role are initially bound to objects given as parameters. Note that in the scope of an alliance type role names can be used as both type specifiers (as, e.g,. in the formal parameter declaration of birth) and variables (as, e.g., in the **assoc**-statements). The clock-role is bound to a newly created object of type Timer[3]. A role is bound when the operation **assoc** is executed. The built-in operations **assoc** and its complement **release** are discussed in Section 3.5.

The statement **persistent** denotes that the newly created alliance is to be made persistent. This does not imply that all participants of a persistent alliance must be persistent. Thus, persistence of alliances is treated independently from persistence of objects. In particular, this allows to include transient objects in an alliance. We will return to the issue of persistence in Sections 3.6 and 6.

3.2 Communication Rules

Communication rules map message requests (**on**-clause) guarded by an optional condition (**if**-clause) onto a reaction (**do**-clause). On a first glance, communication rules seem to have

[3] In order to create an object its real type must be known.

```
rules
    var h: Hotel; p: int; d: Date; ok: bool;
    on reserve(d)@client.selHotel if progress = OPEN do
        begin
        reserve(d)@selHotel; progress:= REQ; def(TIMEOUT)@clock;
        end;
    on reply_reserve(ok)@selHotel.client if (progress = REQ and ok) do
        begin
        ack(ok)@client; progress:= BOOKED; undef@clock;
        end;
    on reply_reserve(ok)@selHotel.client if (progress = REQ and not ok) do
        begin
        ack(ok)@client; undef@clock; release(clock); timer.delete; terminate;
        end;
    on checkin@client.selHotel if progress = BOOKED do
        begin
        checkin@selHotel; progress:= CI;
        end;
    on pay@client.selHotel if progress = CI do
        begin
        pay@selHotel; progress:= PAID;
        end;
    on checkout@client.selHotel if progress = PAID do
        begin
        checkout@selHotel; terminate;
        end;
    ...
```

Figure 4: Specification of communication rules

much in common with event-condition-action rules of active database systems. But they differ in how they are raised, in how they are evaluated, and in the scope of their visibility.

The definition of a message request consists of an optionally parameterized message (the expression before '@') and two role names. The first role name denotes the origin of the message request, the second role name the receiver of the message. The second role specification is necessary since there may be more than one role which can receive a certain message in a multi-party alliance. Message requests are exclusively visible to the alliance to which the message originator has been bound via the first role. In contrast, in active database systems events usually are globally visible.

A communication rule can only "fire", i.e., the specified reaction is executed, if the given condition evaluates to true. Thus, the code in the do-clause of the first rule in Figure 4 is only executed when the specified message request of the client has occurred and the variable progress contains the value OPEN. An **if**-clause is restricted to a boolean expression over state variables and message parameters given in the **on**-clause of the same rule.

On detection of a message request an alliance may react by modifying some local state variables, and/or by indicating messages at roles, and/or by terminating (**terminate**). Termination means that the alliance is removed from the system and no further message requests are handled (the "connection is closed"). In the first rule of Figure 4 the alliance indicates a message reserve(d) at role selHotel in order to deliver the reservation request of the client. In addition it sets variable progress to REQ and indicates a second message def(TIMEOUT) at role clock.

Since alliances have their own memory and can indicate arbitrary messages at roles — provided they are declared in the roles-clause — they can map message requests at sender roles onto message indications at receiver roles in a manner similar to adaptors [33]. The second rule of Figure 4 shows a very simple one-to-one mapping: a reply_reserve(ok) message at role selHotel is mapped onto an ack(ok) message at role client. More complex mappings are possible, too. Additionally, alliances can react to a single message request by indicating an arbitrary number of new messages and, therefore, can suppress messages, can notify a third party about a communication without hard-wiring such notification schemes into objects, or can multicast messages, as we will discuss in more depth in Section 3.4.

The **rules**-clause may contain local variables as is the case in Figure 4. In contrast to state variables their bindings are only valid in the scope of one rule execution.

If we briefly compare alliances with current active database systems we observe that the function of alliances must be simulated there by objects. In the alliance model the underlying execution model recognizes the sending of a message, whereas in active database systems events are implicitly associated only with begin and end of method execution. Consequently, the simulating object can at best register the message but has no control over it.

3.3 Handling Constraint Violations

The rules in Figure 4 are a first implementation of the dynamic model of the cooperation given in Figure 2. We mapped the messages *reject* and *ack* onto a single parameterized message reply_reservation where the parameter value false indicates a rejection and true an acknowledgment by the hotel. We left out the iteration between states *OPEN* and *REQ* which we will add later.

```
on reserve(d)@client.selHotel if progress ≠ OPEN do;
on reply_reserve(ok)@selHotel.client if progress ≠ REQ and ok do cancel@selHotel;
on reply_reserve(ok)@selHotel.client if progress ≠ REQ and not ok do;
on alarm@clock if progress = REQ do
    begin
    ack(FALSE)@client; release(clock); timer.delete; terminate;
    end;
on alarm@clock if progress ≠ REQ do;
on checkin@client.selHotel if progress = OPEN or progress = REQ do
    signal("checkin@client.selHotel")@client;
on checkin@client.selHotel if progress = CI or progress = PAID do;
on pay@client.selHotel if progress ≠ CI do signal("pay@client.selHotel");
on checkout@client.selHotel if progress ≠ PAID do remind@client;
```

Figure 5: Communication rules for exception handling

The rules consider only the regular cases. But exceptions may occur as well. For example, objects may request messages when they are not expected to do so, and, thus, may violate constraints on temporal ordering of messages. Take as an example a client object who requests a checkout-message before it sends a pay message. In order to deal with such exceptions one has to apply additional rules (see Figure 5). These rules can be used to hide errors from objects. In order to compensate errors an alliance can raise regular message requests at roles. For instance, in the last rule in Figure 5 the alliance indicates a remind message at role client in order to compensate the just mentioned error. Note that the erroneous message is not indicated to the selHotel object. Thus, this is can also serve as an example for a message suppression. A special kind of compensation is simply to do nothing at all, as is the case for the first rule of Figure 5.

But in cooperative and distributed environments we have not only to deal with unexpected message requests but also with expected message requests that do not materialize. A well-known technique from distributed systems which can be applied in these cases are timeouts. We used this technique in our example to handle the situation that the hotel does not reply to a reserve message within a predefined time (TIMEOUT). The alliance uses an object of a special built-in type Timer which raises an alarm event to signal a timeout. Timers differ from regular objects in that they can raise events directly with alliances, but can be viewed as normal objects in all other ways. Of course, besides timeouts other well-established techniques like reindicating a message can be implemented.

All exception handling mechanisms considered so far are intended to "repair" an abnormal situation in the course of a cooperation in a way that is transparent to the objects. If a designer decides that this transparency may not be appropriate in a certain situation he or she may resort to *signals*. Signals are special events which make errors visible to objects. Some examples are given in Figure 5. Signals will only have an effect if objects can detect them and react to them (cf. Section 5). Note that alliances do not make any assumptions about the future behavior of objects after a signal has been raised with them.

3.4 Set-Valued Roles

Before a client tries to reserve a hotel he or she normally has to select one from a given set of hotels. As selection criterion one may choose the price of a room. Such a query normally consists of a "price"-message to all hotels in the given set and a reply message of all hotels back to the client. This situation is well-known in both the telecommunications and the database community. In the first such kind of communication pattern is known as multicast. In the latter one would call it a simple query against a set of objects.

In the alliance model queries are supported by set-valued roles. Figure 6 shows a fragment of an alliance type which implements the price-query. A single-valued role client communicates with a set-valued role servers and vice versa. In the case that a message request is directed to a set-valued role the alliance will indicate the message to *all* members of the role — provided the request is legal. The first rule in Figure 6 "multicasts" a price-request by a client to all servers.

The second and third rules of Figure 6 show how to deal with message requests from a set-valued role. Here we are interested in each single communication. Therefore, a request event is raised with the alliance every time a member of a set-valued role requests a message. The **from**-clause allows to refer to the originator of the message. In Figure 6 the alliance collects all responses into a state variable prices. It also guarantees that only one reply of each server object is considered (in our example the first reply). After all servers have responded (third rule in Figure 6) the alliance indicates the result to the client object.

Note that a designer can define arbitrary "query protocols". For instance, the query protocol of Figure 6 may be extended by timeout mechanisms as we already did with the

```
alliance PriceQuery is
roles
    client: ← [price({[ANY; int]})]; → [price];
            !! ANY denotes an arbitrary object, {[ ]} a set of tuples
    servers: ← {price}; → {reply_price(int)};
states
    price_req: bool; prices: {[ANY; int]}; replied: {ANY}; timer: Timer;
birth (c: client, s: servers) is
    begin
        assoc(client, c); assoc(servers, s); assoc(clock, timer.create);
        price_req:= FALSE; prices:= EMPTYSET; replied:= EMPTYSET;
    end;
rules
    var i: ANY; p: int;
    on price@client.servers if not price_req do
        begin
            price@servers; price_req:= TRUE; def(TIMEOUT)@clock;
        end;
    on reply_price(p)@servers.client from i
        if price_req and not i in replied and replied.size < servers.size − 1 do
        begin
            prices.insert([i, p]); replied.insert(i);
        end;
    on reply_price(p)@servers.client from i
        if price_req and not i in replied and replied.size ≥ servers.size − 1 do
        begin
            prices.insert([i, p]); price(prices)@client; timer.delete; terminate;
        end;
    !! communication rules for exception handling
end alliance PriceQuery;
```

Figure 6: Query implemented as alliance

hotel reservation example. This could be used to indicate partial results of a query when a predefined time limit is exceeded, which might be a better solution in some cases than indicating nothing until the last object has answered, especially if the set of queried objects is large and/or distributed over a network.

3.5 Dynamic Role Bindings

The designer of the cooperation protocol given in Figure 2 considered a possible rejection of a reservation and, therefore, decided to iterate through the states *OPEN* and *REQ* as long as the requested hotel did not acknowledge a reservation. But this iteration only makes sense if another hotel can be chosen for every request. Consequently, we have to allow that role bindings can change over the life-time of an alliance.

In order to modify role bindings we provide two built-in operations **assoc(**⟨role name⟩, ⟨object identifier (oid)⟩**)** and **release(**⟨role name⟩**)**. These operations are applicable to both single-valued and set-valued roles. **assoc** assigns the given object to the specified role (in the case of a set-valued role the object is inserted into the role set), **release** discards the actual role binding (in the case of set-valued roles the bindings for the whole set are discarded). For set-valued roles two additional "overloaded" operations are provided: **assoc(**⟨role name⟩, ⟨oid set⟩**)** that allows to bind a whole set of objects to a role, and **release(**⟨role name⟩, ⟨oid⟩**)** which discards the binding of an individual object.

The objects to be bound to a role can be passed to an alliance as parameters of the **birth**-operation or as message parameters. Figure 7 shows the modifications of the rules of Figure 4 which are necessary to implement the above mentioned iteration and the replacement of requested hotels. An additional parameter containing the newly selected hotel has been added to the message reserve. The object passed as parameter is bound to the role selHotel on a reserve request, and the binding is discarded on rejection by that hotel.

```
on reserve(h, d)@client.selHotel if progress = OPEN do
    begin
        assoc(selHotel, h); progress:= REQ; reserve(d)@selHotel;
    end;
on reply_reserve(ok) if progress = REQ and not ok do
    begin
        ack(ok)@client; undef@clock; progress:= OPEN; release(selHotel);
    end;
```

Figure 7: Dynamic role bindings

Assoc establishes a physical connection between an alliance and a newly bound object. For this the object is localized in the given (potentially heterogeneous) address space and a so-called association control event *assoc* is raised with the objects.

Release destroys the physical connection between an object and an alliance. For this, it raises a second type of association control event — *release* — with the object whose binding shall be discarded. Finally, it should be noted that termination of an alliance automatically releases all participants.

Note that the global universe of objects and alliances may be physically distributed.

Consequently, keeping the global structure of objects and alliances consistent[4] may require updates at several nodes. The association control events indicate when such updates are necessary.

3.6 Execution Model

This leads us to the question of how communication rules are evaluated. Whenever the specified message request of a rule has occurred and its condition is satisfied the specified action is performed atomically. Atomicity means that either all messages are indicated and all updates of state variables are executed, or none. Moreover, if an alliance is persistent the new state will survive potential system failures. Whether the message indications survive the situation depends on the persistence of objects with which they have been raised.

In order to prevent anomalies caused by intra-alliance concurrency without implementing expensive synchronization mechanisms, we enforce sequential ordering of actions, i.e., while a rule is evaluated newly requested messages are buffered. This makes sense because an alliance is not a resource shared by competing partners but a service regulating their interaction. If an alliance is persistent the messages must be buffered on durable storage.

Conceptually, an alliance performs the following steps in an infinite loop. It determines the rules for which the specified message request has occurred. If several rules qualify, it indeterministically selects one for execution. In the context of one alliance we allow concurrency between objects bound to different roles but we forbid intra-object concurrency in the context of a single role. In other words, we assume a total ordering of events at one role, but assume no ordering of events across different roles. Always the smallest event — with respect to the aforementioned ordering — in the history of a role is selected (FIFO strategy).

After the execution of the selected rule has terminated the requested message is discarded.

4 Alliances in Distributed Systems

Modern applications are distributed. Consequently, it is only natural to assume that the objects participating in an alliance are spread across several nodes of a network. A premise of our work is that distribution should be treated as an add-on feature to object systems,

[4]This structural consistency may be termed as referential integrity between objects and alliances.

without affecting the local behavior of objects. In this section we argue that alliances are once more the ideal place to embody the necessary application-specific regulations governing distribution [24], provided we can do so transparently to the objects so that the developers of objects can concentrate on their functionality proper.

4.1 Distribution Policies

For the following we assume that even in a distributed object system each object resides in its entirety at a physical node in the network. Under this assumption the conventional object paradigm seems to raise no obstacles to distribution. Because objects encapsulate their information and, hence, have no state in common they interact solely via message exchanges, which are easily mapped to the physical messages in a distributed system.

One straightforward solution to the implementation of an alliance in a distributed world is to maintain it as a physical entity, much like an object, and hence have it reside at a single node. The necessary support for the execution model of alliances (cf. Section 3.6) must be replicated at each node, essentially by distributing the event interfaces of Figure 1. Note that in this solution sending a message from one object to another now could involve up to three different nodes: the ones hosting the caller, the callee and the intermediary alliance. This added traffic may degrade quality parameters like performance or reliability. Distributed systems can counter such degradation by mechanisms like replication [16] or object migration [17], and may even add new qualities. Additionally, these mechanisms are the weapons to counter the anomalies that are the only reason why developing distributed systems is a task far more complex than realizing equivalent centralized applications.

Putting these mechanisms to good use is the matter of a *control policy*. Much like a collaboration policy is expressed by some sort of protocol inside an alliance, the policy of how to control distribution should be kept strictly a matter of the alliances, invisible to the objects or, in other words, encapsulated within alliances. As an added benefit, the code that manages collaboration contexts and distribution policies is concentrated in one place and thus easier to develop and maintain.

The remainder of this section will use one of the aforementioned mechanisms to illustrate the power of alliances for distribution policies. We choose the mechanism of object migration for three reasons. Firstly, object migration is itself a meaningless concept unless it is accompanied by a proper policy. Secondly, mobile entities in distributed systems are nowadays found in numerous systems, as shown in a comparative study [28]. Thirdly, object migration is technically simple, so that our discussion need not be filled with extensive discussions on the trade-offs of various flavors of the mechanism which would detract from the policy proper.

4.2 Mobile Objects

Since there is no distinction between local and remote object invocations, it is possible to move objects at runtime among the nodes of a distributed system. Technically, one needs an additional level of indirection to trap remote invocations and forward them to the current location of the remote object. One also depends on location-independent object identifiers and a mechanism for locating migrated objects. The technical details of these mechanisms are well understood (see, e.g., [17]) and need no further discussion.

As mentioned before, migration of objects should be subjected to a control policy. Such a policy depends on what anomaly should be countered. In a distributed world such counter measures aim at *load sharing* to take advantage of lightly used computers, at improved *communication performance* in bringing interacting objects together to reduce the communication cost or at *availability* in moving objects to different nodes to provide better failure resilience[5].

Even though this is a small list, one can clearly identify conflicts among the goals. Note, for example, that availability calls for dislocating objects, while performance calls for colocating them. What primary goal is to be followed is subject to the stated policy.

4.2.1 Controlling Migration — the Conventional Approach

Linguistic support for mobile objects normally comprises means to **fix()** objects to nodes, to **migrate()** objects directly to a target node or target object, and to keep communicating objects permanently affixed to one another by issuing an **attach()** among them. All those primitives are based on two implicit assumptions:

Objects know their future communication patterns. If this assumption does not hold, there is no basis for any migration decision. Hence, an object should know its communication partners and how the cooperation with them will develop.

[5]A more comprehensive discussion of possible goals could be found in [17]. We selected the items which are commonly regarded as being of general importance.

Primitive	Semantic
migrate(O_1, O_2)	Migrates O_1 to the current location of O_2
location_of(O)	Returns a node object denoting the current location of O
is_resident(O)	Returns **true** if O is at the local node
attach(O_1, O_2)	A–transitively A–assigns O_1 to O_2 together with a simultaneous **migrate(O_1, O_2)**
detach(O_1, O_2)	Break an attachment

Figure 8: Primitives to express policies inside alliances.

All objects are trusted. Any object may call for arbitrary attachments or fixings of other objects. Hence, no object can exert sole control over what other objects it is attached to or whether it is fixed at the moment. In order to make sure that all policies expressed by individual objects sum up to a sensible behavior, all objects are assumed to behave in a reasonably fair way.

These assumptions appear enforceable for monolithic distributed applications that are set up by a single programmer or a small, closely-knit design team. In a world of autonomously developed objects which cooperate only on a case-by-case basis such a condition may not hold any longer. For example, objects will underestimate the effect of an **attach** as they are not informed about the transitive attachments of other objects.

4.2.2 Expressing control policies through alliances

If we wish to entrust alliances with responsibility for distribution, the aforementioned linguistic primitives must come under their sole control. Figure 8 shows the primitives to be used by alliances to control migration. The semantic of the primitives needs some modification from the conventional approach, though, because the should exploit the knowledge what objects are working together on a common task and the knowledge when the common task ends. Hence, **attach()** is defined in terms of two properties, A–transitive and A–assigns. A–transitive means that transitive attachment is defined only within a given alliance, i.e., that attachments defined by different alliances are never combined. Thus, alliances have a full understanding of the consequences of their attachments. A–assigns limits the attachment to the lifetime of the alliance. As soon as an alliance terminates, all attachments defined by it are dissolved. **is_resident()** has been included although its functionality could also be expressed via the **location_of()** primitive. The reason is that the test whether an object resides on the same node as the alliance or not can always be computed solely on the basis of local information, whereas one generally depends on remote information to enquire the current location of an object. Further, no **fix()** primitive is defined because its effect can be obtained by combining a **location_of()** that returns a node object *NO* with a subsequent **attach(O,** *NO*) primitive. In this way the more limited A–transitive semantics of attach is enforced for the fixing.

```
birth(c: client; h: selHotel) is
    var h: selHotel;
    begin
        assoc(client, c); assoc(selHotel, h); progress:= OPEN;
        attach(self, c); persistent;
    end;
...
on checkin(d)@client.selHotel if progress = BOOKED do
    begin
    attach(client, selHotel); attach(self, selHotel);
    checkin(d)@selHotel; progress:= CI;
    end;
```

Figure 9: Expressing distribution policies through alliances

Alliances allow full control of a cooperation and distribution policy including their interdependencies, without affecting other cooperations and with no influence by other cooperations. In addition, alliances are a tool to cope with conflict resolution between contradictory attachment requests. Without any knowledge of the various collaborations that gave rise to the conflicts the underlying system has no way to weigh them in order to resolve them. Given the alliances, the system may now associate conflicts with collaborations, and fine-tune its resolution strategies. For example, since the scope of attachments is limited by the lifetime of the alliances it may impose an order on conflicting attachment requests. Alternatively, it may collect statistical data on the activities associated with an alliance in order to estimate future behavior, and use the predictions for more sophisticated decisions; e.g. to give privileges to a request issued by a collaboration that currently accounts for the biggest part of the overall activity.

Figure 9 modifies the **birth** operation and communication rules of Figure 4 to realize a simple distribution policy for our hotel example. At instantiation time the alliance is attached to the client object, as each communication rule encompasses a message to or from the client. When the client performs a checkin on the hotel both client object and alliance migrate to the hotel object.

Again the discussion clearly demonstrates the superiority of the alliance approach over a world composed of objects only. In the latter all migration strategies are spread across the encapsulated method implementations of numerous objects, so that it is difficult to see how conflicts can be recognized, let alone be dealt with according to flexibly varying strategies.

5 Integration with an Object Model

Alliances enforce the collective behavior of a collection of objects in a distributed environment. Objects are the instigators of all actions within alliances, and in the vast majority of cases they are also the carriers of all major actions if we presume as usual that the majority of information processing takes place locally. As an example object model we briefly introduce our model of autonomous objects which is based on preliminary work [19] and show how alliances can be integrated with it.

An autonomous object has all the qualities of a traditional object, i.e., a logical object identity, an interface which defines a set of messages it can receive, and an implementation which consists of a structure, i.e., a set of attributes, and operations that implement the reaction to messages. In addition, an autonomous object possesses a set of guards which implement its synchronization constraints. Once an autonomous object has been created, it has its own thread-of-control to evaluate the guards according to a system-defined execution model (we omit the details because they are not important in the context of this paper).

As is standard, autonomous objects communicate with each other by message passing. However, since in cooperative environments each object may take the initiative for a communication, and an object may be involved in more than one cooperation simultaneously, we presume an asynchronous message transfer instead the traditional synchronous message passing corresponding to a procedure call. (This explains the one-way communication of Figure 1).

Questions we have to answer from an objects' viewpoint are how alliances are created, how inter-object messages are mapped onto events at roles, how the context of a received message, i.e., a role, could be derived in order to allow objects to control concurrent alliances, and how association control events and signals affect objects.

An object can create an alliance explicitly by calling the **birth**-operation. It can pass initial participants as parameters (cf. Section 3). The **birth**-operation returns a handle to the newly created alliance. The object may store it in its local state. For future message sendings the object can refer to this handle together with a role specifier in order to address the message receiver (e.g., myAccommodation("selHotel").reserve(d), where myAccommodation is a variable containing the handle to an alliance of type HotelAccommodation). Alliance handles can be interpreted as generalization of reference variables (i.e., attributes that point to objects).

Since we allow that one object can be bound to more than one role of an alliance, it is sometimes necessary to specify a second role to unambiguously identify the location of a message request (e.g., myAccommodation("selHotel").reserve(d) as client).

In order to allow a receiver of a message to identify its context each message is provided with pre-defined parameters that contain the necessary context information, i.e., handle to alliance, role where the message has been indicated, and role of the sender object. These context parameters need not explicitly be declared.

All message requests along the execution of an operation are by default associated to the role where the message which caused the execution of this operation has been indicated unlike the object's implementation specifies it otherwise. For instance, when a hotel object h receives a reserve(d) message as selHotel in an alliance a, all messages that h requests throughout the operation which is executed as reaction on this message are by default raised at selHotel of the same alliance a.

An object need not concern about association control events. If — for some reasons — an object need to know whether it is actually bound to a certain role or not it can dynamically check its actual role bindings[6].

Objects can "catch" signals in the same way they receive regular messages. Of course, no sender object is provided.

So far we have sketched a solution which allows objects to exploit knowledge about their participation in alliances, particularly, to synchronize concurrent messages if they do so in several ones simultaneously, and to establish new ones. Although objects refer to alliances

[6]Such a check can be compared to a NULL- or nil-test in classical object models.

explicitly there remains much leeway for a designer to specify or adapt customized protocols without necessitating reimplementation of objects. Consider, as an example, the query protocol in Section 3.4.

On the other side this solution does not meet our requirement in Section 1 on objects which can remain completely oblivious in t he participation in an alliance. Hence, as part of our ongoing work we investigate under which conditions message control across alliances can unambiguously be derived from the conventional format of message requests.

6 Prototypical Architecture

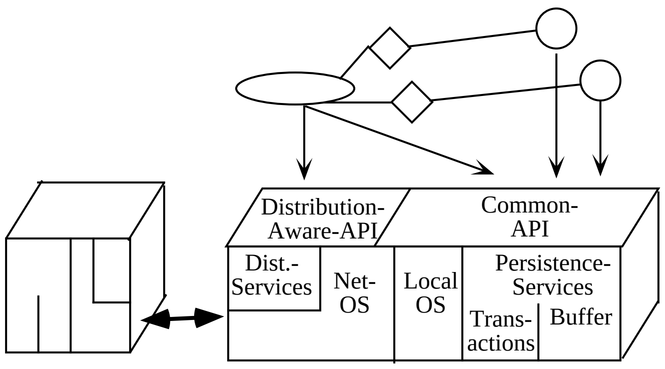

Figure 10: Prototype Architecture

Our current prototypical architecture is depicted in Figure 10. It is built around services on each node, which comprise persistence, capabilities of both a local and a network operating system, and specialized distribution mechanisms. Objects (circles in Figures 10) and alliances (ovals in Figures 10) rely on two APIs to use those services. The first one is the *Common–API* that provides persistent memory, transactions, local input/output and distribution-independent message passing between objects. The second one, the *Distribution–Aware–API*, realizes the distribution mechanisms subject to the distribution policies that are coded inside the alliances. Hence, alliances use both APIs, while objects are only mapped onto the Common–API. Even though objects cannot use distribution mechanisms directly, they are subject to distribution policies. As an example, the location-independent communication has to work regardless of the current location to which objects have migrated via the

Distribution–Aware–API.

Currently, we have built two prototypes to validate our approach. The first one was integrated with ObjectStore [20] as a provider of persistence services, but without putting the system into a distributed world. All communication is performed via a global object base, i.e., all objects and alliances are database objects which must be modified to request and indicate messages along actions which are executed as transactions. This is an easy-to-implement but inefficient solution for both intra-node and inter-node communication between objects and alliances.

The second one was used to look into our approach of specifying distribution policies inside alliances and bases on OSF/DCE as the net–OS and on DC++ [30] as the provider of distribution mechanisms, in particular the capability to migrate objects dynamically. One major problem with this second prototype is to preserve the ordering of causally dependent events between an object and an alliance at one role. This could trivially be achieved in the first, transaction based prototype — at the cost of performance.

The architecture does not impose severe restrictions on how to map objects and alliances onto runtime–incarnations. If we used a base system that allows for replication or other specialized distribution services, it might have been sensible to map an object or alliance onto multiple incarnations. Consider as an example the use of virtual synchronity, provided by the ISIS ABCAST mechanism [4] as the communication base, added with aggressive replication of the alliance on the node of each object bound to the alliance. All those alternative approaches allow for optimizations based on the specific capabilities of the underlying distribution services, and offer possibilities similar to those discussed for migration in Section 4.

The foremost objective of our prototypes is to demonstrate the utility of our approach for a number of applications. One is simultaneous engineering. A second scenario with a much larger degree of distribution that recently has become available to us, distributed truck fleet control, is based on a project from distributed artificial intelligence.

7 Conclusion and Outlook

In order to meet the requirement for communication abstractions in object-oriented models we have introduced alliances which allow for the separation of cooperation and distribution aspects from the objects, and concentrate them in a separate construct.

The participants of alliances are autonomous objects which cannot always be expected to meet their obligations in a cooperation. Alliances can compensate protocol violations by indication of compensating messages with objects. Alliances describe long-lived cooperations between a large set of objects and with changing participants. For this we allow alliances to be persistent with persistence of alliances treated independently from that of objects. We further introduced set-valued roles which support customized query protocols in a cooperative environment, and dynamic role bindings which allow to dynamically associate objects with alliances and release them if they are not needed any longer.

We further assumed that cooperation takes place in a distributed environment. We demonstrated how alliances can be used to implement customized distribution policies, using object migration as an example. Those policies need neither to be "hard-wired" into objects nor had we to rely on global information which is rarely available in large information systems.

We finally showed how alliances can be integrated with an object model and outlined our prototypical architecture.

By going beyond similar recent approaches, and in particular by adding persistence as an issue central to object bases, we moved into uncharted waters, thus raising a good number of novel questions. We mention two of them.

In order to understand a cooperation protocol a more abstract way of specification seems appropriate. It should be restricted to a declarative specification of multi-object constraints and should not embody the specification of how these constraints are enforced. Several approaches propose some kind of temporal logic for this purpose (e.g., [25]). We presently investigate how temporal-logic-based specifications can be mapped onto alliances in a systematic manner and how alliances types can be verified against those specifications. The second question arises from the relegation of distribution policies to alliances. Since alliances may share objects distribution policies of different alliances may lead to conflicts. We have begun to develop a simulation model for alliances in order to experiment with various conflict resolution strategies.

References

[1] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, Sep 1990.

[2] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 3–21. MIT Press, 1993.

[3] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. of 16th Intl. Conf. on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994. IEEE.

[4] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, Dec. 1993.

[5] A. P. Buchmann. Modeling heterogeneous systems as an active object space. In *Proc. of 4th Int'l. Workshop on Persistent Object Systems*, Martha's Vineyard, Sep 1990.

[6] Intelligent agents. Special Edition of Communications of the ACM, July 1994.

[7] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, Santiago de Chile, Chile, Sep 1994.

[8] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M.-C. Shan. Third generation tp monitors: A database challenge. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 393–397, Washington DC, May 1993.

[9] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: A uniform approach. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991.

[10] S. Gatziu and K. R. Dittrich. Samos: An active object-oriented database system. *IEEE Quarterly Bulletin on Data Engineering*, Jan. 1993.

[11] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 327–336, 1991.

[12] J. Gray and A. Reuter. *Transaction Processing: Concept und Techniques.* Morgan Kaufmann, New York, 1993.

[13] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proc. of ECOOP/OOPSLA*, pages 169–180, 1990.

[14] I. M. Holland. Specifying reusable components using contracts. In O. Lehrmann Madsen, editor, *Proc. ECOOP'92*, LNCS 615, pages 287–308, Utrecht, The Netherlands, 1992. Springer-Verlag.

[15] International Organization for Standardization (ISO). *Information Processing Systems — Open Systems Interconnection — Reference Model*, 1984.

[16] P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, Englewood Cliffs, 1994.

[17] E. H. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, Feb. 1988.

[18] D. G. Kafura and K. H. Lee. Inheritance in actor based concurrent object-oriented languages. In S. Cook, editor, *Proc. ECOOP'89*, British Computer Society Workshop Series. Cambridge University Press, 1989.

[19] A. Kemper, P. C. Lockemann, G. Moerkotte, and H.-D. Walter. Autonomous objects: A natural model for complex applications. *Journal of Intelligent Information Systems*, 3(2):133–150, 1994.

[20] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore database system. *Communications of the ACM*, 34(10):50–63, Oct 1991.

[21] L. Liu and R. Meersman. Activity model: Declarative approach for capturing communication behaviour in object-oriented databases. In *18th International Conference on Very Large Data Bases*, pages 481–493, 1992.

[22] P.C. Lockemann and H.-D. Walter. Activities in object bases. In N.W. Paton and M. H. Williams, editors, *Rules in Database Systems (Proc. of the 1st Int. Workshop on Rules in Database Systems)*, Workshops in Computing, pages 3–22. Springer Verlag, Sep. 1993.

[23] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[24] Nancy A. Lynch. A hundred impossibility proofs for distributed computing. In *Proc. 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–27, Edmonton, Alberta, Canada, August 1989.

[25] A. HH. Ngu, R. Meersman, and H. Weigand. Specification and verification of communication constraints for interoperable transactions. *International Journal of Intelligent and Cooperative Information Systems*, 3(1):47–65, 1994.

[26] O. Nierstrasz. Regular types for active objects. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, volume 28 of *ACM Sigplan Notices*, October 1993.

[27] M. H. Nodine, N. Nakos, and S. B. Zdonik. Specifying flexible tasks in a multidatabase. In *Proc. of 2nd Int. Conference on Cooperative Information Systems*, pages 3–14, Toronto, Canada, May 1994.

[28] M. Nuttall. Survey of systems providing process or object migration. Technical Report Imperial College Research Report DoC 94/10, Imperial College, London, UK, 1994.

[29] The Object Management Group Inc. *The Common Object Request Broker: Architecture and Specification*, OMG document no. 93.12.1. revision 1.2 edition, 1993.

[30] A. B. Schill and M. U. Mock. Dc++: Distributed object–oriented system support on top of osf dce. *Distributed Systems Engineering Journal*, 1(2), 1993.

[31] J. van den Bos and C. Laffra. Procol: A parallel object language with protocols. *ACM SIGPLAN Notices, Proceedings OOPSLA'89*, 24(10):95–102, Oct. 1989.

[32] H. Wächter and A. Reuter. The contract model. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–264. Morgan Kaufmann, 1992.

[33] D. M. Yellin and R. E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 176–190, Portland, Oregon, USA, Oct. 1994.

[34] A. Yonezawa, editor. *ABCL—An Object-Oriented Concurrent System.* The MIT Press, 1990.