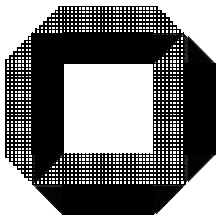
Object-Oriented Protocol Hierarchies for Distributed Workflow Systems

P. C. Lockemann, H.-D. Walter

Interner Bericht 42/95



Universität Karlsruhe

Fakultät für Informatik

(Submitted for publication)

Abstract

Distributed software systems such as groupware and workflow systems will play a key role in the near future. While numerous models which promise highly sophisticated functionality are proposed in the literature their implementation is still a difficult and very expensive task. Therefore existing systems fall far behind their promises.

Entities of the workflow level are often autonomous. Consequently, they are related to each other in more than a fixed client/server configuration: they often perform their activities in collaboration. Workflow models also contain a lot of information about the system's dynamics. If one uses objects as an implementation model — which is the most preferable of all possible choices today — all these aspects must be mapped onto the same abstraction which is mainly concerned with functionality of entities but neglects relationships almost completely, i.e. treats entities in isolation. The dynamics is hidden inside object implementation, including and especially those concerned with cooperation.

We therefore propose an extended object model which allows the definition of dynamic relationships, called *alliances*, between objects. Alliances define and enforce cooperation protocols at the object level. The semantically enriched object model can be implemented on top of common distributed object technology which for its part relies on standard database and communication services. Both an enriched implementation model and the utilization of standard software contributes to the reduction of development costs.

1 Introduction

Future software systems will be distributed running on multicomputer systems[1] where distribution is not only a means for optimization but mainly the prerequisite for offering additional functionality. One example for such kind of software is *groupware* which includes among others *workflow systems*. Workflow systems add computer support to the execution of business or engineering processes. They mainly differ from conventional software in two aspects: First, they are no longer passive tools which act exclusively on users' requests but become proactive systems which invoke operations on their own initiative. Second, the interaction of their components is no longer solely governed by a client-server relationship. Instead, autonomous components cooperate with each other where all components have equal rights. Distribution and autonomy of components imply that cooperation must be communication based[2].

Hence, workflow systems seem to promise users a great deal of functionality. But implementations by far do not keep these promises. Among the reasons responsible for this unsatisfying situation are unsolved technical problems [14]. A number of requirements which are hard to meet make workflow systems difficult and costly to implement. It must be possible to extend a workflow system incrementally because neither its type and its number of workflows nor its users nor its components are completely known a-priori. A workflow system must be designed around autonomous components because these need the autonomy to be able to optimally perform their tasks. As stated in the beginning, workflow systems must run in distributed multicomputer environments where not only distribution but also heterogeneity of hardware and system software has to be considered. Since numerous standard software which solves problems of distribution and heterogeneity are available today or will be available in the near future (e.g., database standards as [10] or distributed system standards as [42, 41]) it is essential that workflow systems use these standards as far as possible in order to reduce development costs, keep the system open for future extensions, and enhance its portability. The components of a workflow system must be interactive because they can solve their tasks often only if they cooperate with human users or other software components. Autonomy, interactivity, and distribution imply that components should be pro-active and the workflow system internally concurrent. Finally, due to the autonomy the temporal order of arising situations and possible executions of processes are unknown at specification time. Consequently, a workflow system must be able to react flexibly to situations, which is only possible if one avoids a strictly procedural description and specifies the interaction in a more declarative manner.

These requirements — especially distribution, autonomy, and interactivity of components — suggest to use object-oriented technology to implement workflow systems. This would lead to a three-layered architecture which would implement workflow systems on top of distributed object management systems as, e.g., *CORBA*-based systems, which rely on standard services. Unfortunately, today's distributed object management systems consider communication based cooperation only in the narrow request-reply context (e.g., RPC). High-level cooperation contexts as workflows are not expressible. This leads to considerable programming effort in order to close the gap between workflow and object models.

Cooperation rules play a key role in the task to bring objects together. It has been recognized for years that it is very difficult to express these cooperation rules such as, e.g., temporal ordering of messages between a set of objects, especially if objects are autonomous (i.e., may act in an unforeseen way) and have been implemented unaware of the cooperation contexts they are used in. In essence, expressing multi-object constraints in today's distributed object management systems can only be done by "hardwiring" them into the object implementation and, for that matter, spreading 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 difficult to control. It further obstructs reusability of objects — a strength often claimed for the object-paradigm. A programmer must also anticipate all possible "misbehaviors" of cooperation partners which is an unsolvable task in complex environments.

Therefore, we propose to enrich the object layer by a novel construct called *alliance* which defines a dynamic relationship between cooperating objects and materializes cooperation protocols. This construct extends comparable approaches in the literature, e.g., [3, 4, 22, 33, 34, 52] — which are mainly intended to bridge type incompatibilities of two communicating objects or extend interfaces of server objects

[1] We use the term multicomputer to refer to all systems with interconnected autonomous processors with their own memory [48]. Both tightly coupled systems as transputers and hypercubes and computer networks as LAN and WAN belong to this category. For the applications discussed in this paper, computer networks are the prevailing multicomputer architectures.

[2] This is in contrast to "shared-memory" based cooperation where we mean "shared-memory" in its broadest sense.

by protocols which constrain sequences of possible client invocations — by supporting large evolving sets of cooperating objects, long-living tasks, compensation of errors, and integration into a distributed environment.

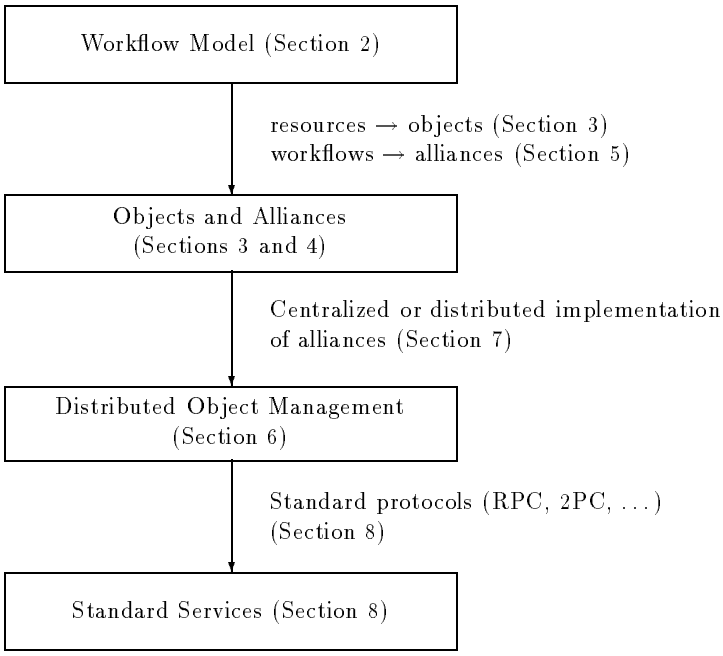


Figure 1: Layers of a distributed workflow system

As a consequence we can view a workflow system as a stack of four layers as shown in Figure 1. A workflow model which defines activities, resources, and dependencies between activities is mapped onto an object-alliance model. Objects and alliances on their part use common distributed object technology which is based on standard services.

The outline of this paper is as follows (cf. Figure 1). In Section 2 we briefly review workflow systems and their implementation requirements. Section 3 discusses how workflow models can be mapped to objects. Section 4 is dedicated to a novel concept of dynamic relationships between objects. We use these dynamic relationships as implementation model for cooperative workflows. The mapping is discussed in Section 5. Section 6 shows how distributed object management systems can be used to implement "services" offered at the object and alliance level. The implementation of alliances is treated separately in Section 7. Distributed object management relies on standard services, as, e.g., communication services and database services, and their related protocols. Section 8 is dedicated to them. Section 9 discusses advantages and shortcomings of an alliance-based implementation of distributed workflow systems. Section 10 summarizes the main results of this paper, gives a brief overview of our prototype implementation, and sketches ongoing and future work.

2 Distributed Workflow Systems

McCarthy and Bluestein [36] define workflow systems as follows:

> "Workflow management software is a proactive computer system which manages the flow of work among participants, according to a defined procedure consisting of a number of tasks. It coordinates user and system participants, together with the appropriate data resources, which may be accessible directly by the system or off-line, to achieve defined objectives by set deadlines. The coordination involves passing tasks from participant to participant in correct sequence, ensuring that all fulfill their required contributions, taking default actions when necessary."

This definition implies that workflow systems need an integrated view on activities or tasks of a process, e.g., a business process in an office environment or a technical process in a production environment, on actors which do the work, and on required resources, as well as on dependencies between activities. *Workflow models* offer this overall view. A *workflow management system (WFMS)* which coordinates all activities at run-time takes a workflow model as input.

A distributed workflow system is a workflow system where the execution of a workflow may involve several nodes of a network and where the WFMS itself is distributed and not realized as a monolithic server. When we use the term workflow system in the sequel we will always refer to distributed workflow systems if not stated otherwise.

2.1 Workflow Models

A workflow model describes activities, actors, resources, and dependencies between activities [25]. Figure 2 shows an example workflow. It models the processing of a client's order in a transportation company. The company consists of a set of sites which are distributed over the area which is covered by the company (e.g., central Europe or the U.S.). A client turns to its local site (e.g., Munich) with some transportation task (e.g., he or she wants a piano to be carried from Munich to London). The local site, i.e. Munich, is automatically assigned as coordinator in what follows. Depending on the order the coordinator selects a set of relevant partner sites (e.g., Munich, Brussels, Paris, London). All partners simultaneously evaluate the order with respect to their current load, capacity and the order's value. They communicate the result of their evaluation to the coordinator who eventually decides which partner (or in some cases which subset of partners) shall execute the order. Due to unforeseen events, such as truck failures, accidents, or congested highways, the execution of an order must be replanned occasionally. Finally, invoicing and payment concludes the process.

2.1.1 Activities (Workflows)

Activities (or workflows) define what has to be done. Often they are structured hierarchically, i.e., a workflow consists of sub-workflows. The leafs of a workflow hierarchy normally are applications (programs or transactions). Leafs are atomic computational units that are not further divided.

In our example the root workflow "order processing" consists of four sub-workflows "accept order", "planning", "execution", and "invoicing". Workflow "planning" is further divided into two sub-workflows "evaluation" and "assignment".

2.1.2 Actors

Actors define who executes workflows. Actors are assigned to activities dynamically at execution time. Properties usually required from actors are statically defined as *roles*. Roles constrain the set of actors which are qualified to play a certain role in a workflow.

In our example the workflow "accept order" has two roles, *coordinator* and *client*. When this workflow is instantiated at execution time the roles must be bound to concrete instances of actors[3].

In some cases a role describes a set of actors where the cardinality of the set is not known at specification time, e.g., the partner sites in workflow "planning". Set-valued roles are denoted with curly brackets in Figure 2.

Once a workflow is instantiated, assigned actors must be notified. Since actors may be assigned to more than one workflow at a time to-do-lists for actors must be maintained. For instance, a partner site may be concurrently involved in more than one "planning" workflow.

2.1.3 Dependencies

Dependencies between workflows have to do with three issues [25]: execution control, data flow, and logging.

Execution control determines the behavior of a workflow at execution time. The description of execution control can either be procedure-like, or declarative, or trigger-based. In the first case constructs which are well-known from imperative programming languages can be used to specify execution control.

[3] Thus, our example workflow description can be regarded as *workflow type*.

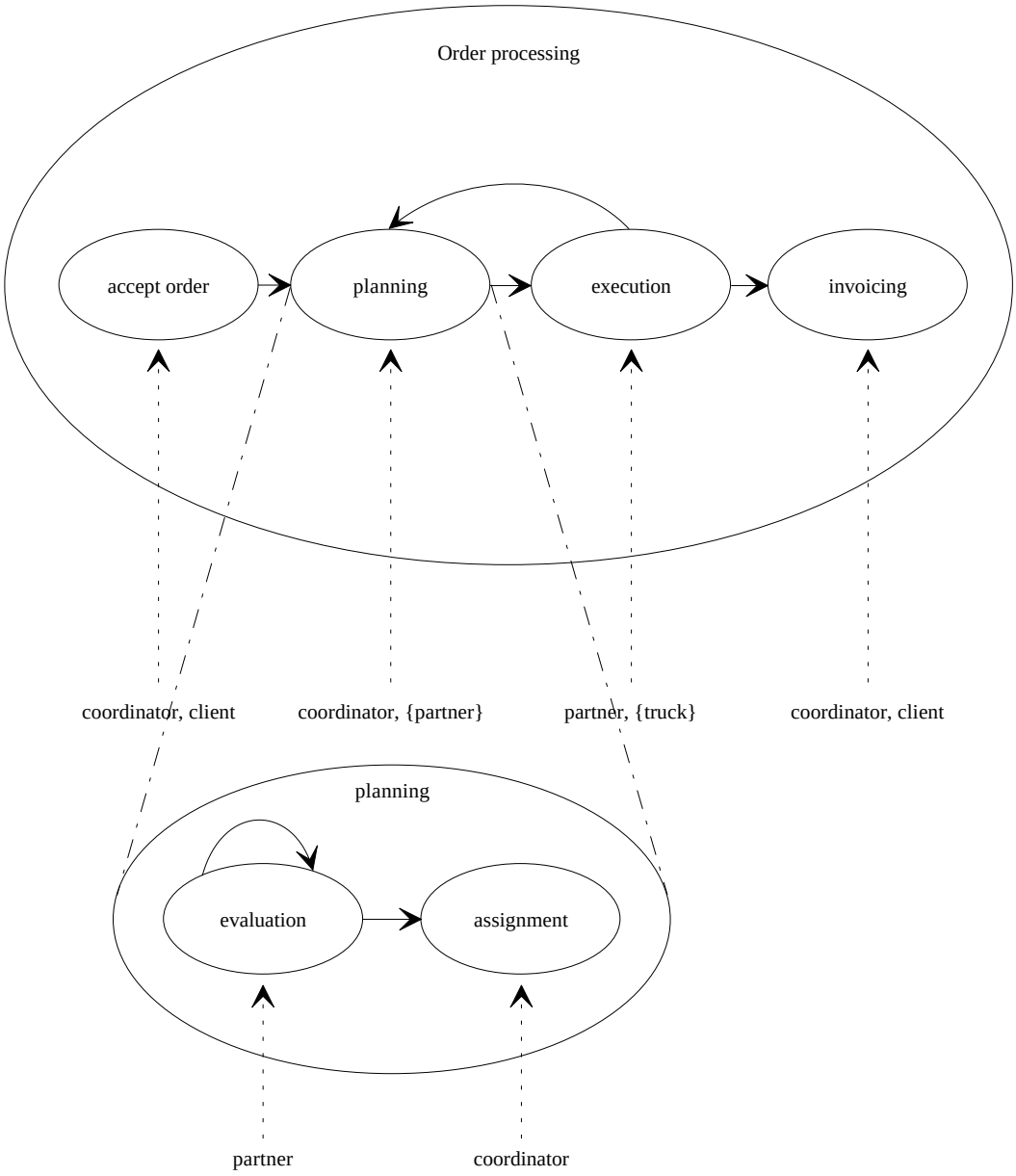


Figure 2: Example workflow

The most important operators are sequential execution (of workflows) $(w_1; w_2)$, conditional execution $(b?w_1 : w_2)$, loops (**while** $b : w$), and parallel execution $(\|w_1, w_2, \ldots)$. Approaches which use this kind of execution control description are among others *Sagas* [17], *ConTracts* [51], *Interactions* [40] which are based on transactions, and, e.g., the commercial workflow product *FlowMark* [31]. The specification of our example workflow in Figure 2 belongs to this class of description techniques, too.

In the declarative case possible executions of workflows are constrained by temporal ordering conditions and existence conditions [7]. Temporal ordering of workflows $(w_1 < w_2)$ expresses that w_1 must take place before w_2 if both w_1 and w_2 take place. Existence condition $(w_1 \Rightarrow w_2)$ denotes that if w_1 takes place then w_2 must also take place (which does not impose any restriction about the sequence in which they take place). In our example one might specify "accept order" < "planning" which allow planning of orders without any order acceptance (e.g., one might think of company internal "pseudo"-orders which are not initiated by a client) and assignment of a partner without any planning (which, e.g., might be

necessary due to time constraints). But if both workflows take place, the constraint must not be violated. It would be further reasonable to demand that "invoicing" ⇒ "execution" which would protect a client against unjustified invoices.

Declarative descriptions of execution control allow users and WFMS to react in a more flexible way on exceptional events than procedure-like descriptions. Of course, the lack of execution semantics must be added at some point later, latest at execution time. Usually a global scheduler is responsible that workflows are executed in a manner consistent with all dependencies [45].

A mix between procedure-like and declarative approaches are trigger-based techniques in the context of active database systems. Trigger-based approaches assume that leafs of a workflow hierarchy are transactions. Execution of transaction can be controlled by way of event-condition-action rules (or active rules) [12] where the detection of an event may lead to the execution of a transaction if some predefined condition over the database state holds. Active rules are either orthogonal to the data model of the DBMS (e.g., [12, 8, 18, 9]), or they are added to objects in object-oriented models (e.g., [13, 20, 5]). Active rules define execution control of workflows on a low level of abstraction. Thus, as the approach proposed in this article, they are better suited as a platform for implementing a WFMS — to which especially their active functionality can contribute — and not so much as part of the interface of a workflow system where users have to specify their workflows by means of active rules.

In large-scale distributed systems trigger-based approaches exhibit severe disadvantages as a means to implement workflow systems. First, active rules are stored globally and independent from a certain application in a database system. Storing active rules globally and independent from applications is fine for their original purpose as a flexible means to ensure database consistency [29]. It is disadvantageous in the context of workflow systems, since it is very hard to extract those active rules which define the dependencies in a certain workflow in a system where many different workflows must coexist, which we expect to be the normal case. Further, it is possible that active rules which define dependencies of different workflows may influence each other which makes it a very hard task to verify whether a set of active rules guarantees some given dependencies. This is also a consequence of the fact that the relationship between active rules and workflows is not made explicit.

A second disadvantage is related to the implementation of active rules systems in distributed environments. We postpone the discussion of this issue to Section 2.3.

The second issue, data flow, has to do with shared data that may lead to dependencies between workflows, too. First, workflows will exchange data among each other, second, different workflows might access common data which might lead to sequentialization of workflows although a user might have defined them as independent.

The last aspect concerning dependencies of workflows is logging of workflow execution which might be used to enhance the reliability of a workflow system.

2.2 Architecture of Workflow Management Systems

Figure 3 shows the conceptual architecture of a WFMS (taken from [25]). A WFMS can be divided into a *kernel* and a *shell*. The first consists of a *controller* which supervises all dependencies between workflows and evaluates the execution control specification. In order to live up to its task the controller relies on various components of the WFMS shell: a pool of software tools (e.g., planning systems and accounting systems) which are part of the workflow system (*tool manager*), a *log manager*, a component which is used to assign actors to roles (*role binding*), a component which offers notification services, e.g., an email system, (*notification*), and a data manager which offers persistence services. Each *actor* has a *to-do list* which contains its tasks to perform (e.g., a list of client orders to evaluate).

Both actors and tools are represented as programs in a workflow system[4]. They differ with respect to their active behavior in the system. While tools are *reactive*, i.e., they only do some work if some actors request it, actors are *proactive*, i.e., can take the initiative to do some work on their own (or on some external input).

[4] Many authors stress that actors include human users or machines, i.e., components external to the system. Since every external component needs an interface to the computer in order to be integrated into a computer-system we assume without loss of generality that every external component is represented as some program in the workflow system.

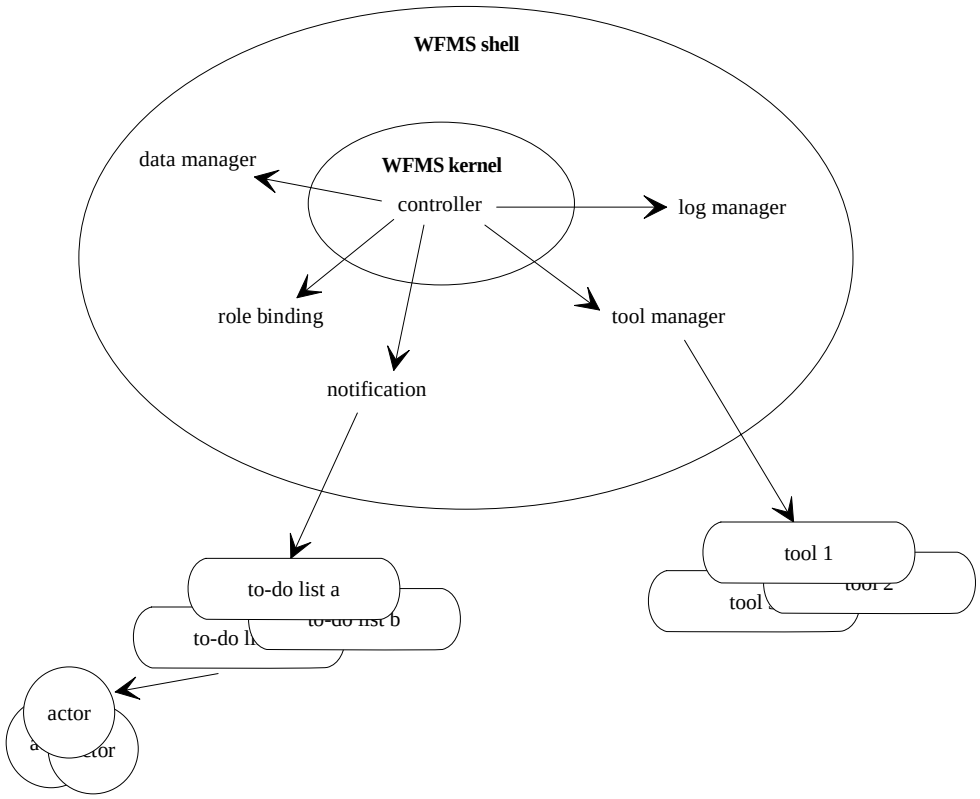


Figure 3: WFMS architecture

2.3 Implementation Requirements

In this section we discuss the requirements a WFMS must meet.

2.3.1 Scalability and Extensibility

Workflow systems must be scalable according to workflows, workflow instances, actors, and tools. A workflow system should not restrict the types of workflows which can be executed. Users should be able to define new workflows and add them to the system whenever this is necessary. Nor should the number of workflow instances which are concurrently active in the system simultaneously be restricted. It should be possible to dynamically extend the workflow system by new actors and tools.

Since today's implementations of WFMS use some kind of client-server architecture (see, e.g., [31]), i.e., the WFMS kernel and shell are realized as servers while actors, tools, and data storages might be distributed over a network, their performance depends on the number of active workflows. The WFMS kernel might become a bottleneck of a workflow system. Consider for instance our fleet control example: execution of workflows at some partner site depend on a WFMS server which might be situated at some other site of the transportation company.

2.3.2 Integration of Autonomous Components

It should be easy to integrate new actors and tools into the workflow system. In order to meet this requirement a workflow system should make only few assumption about these components, i.e., it should consider them as being *autonomous* as far as possible. For example, we cannot expect that software is always developed with respect to its future application in workflow systems (e.g., legacy systems). Another reason to allow components to be autonomous is to give components the necessary latitude to fulfill their highly specialized task in the most optimal way.

Many WFMS are based on distributed transaction systems [45] which implies that the components must offer special functions at their interfaces (e.g., *commit* and *abort* of transactions) and must possess certain internal states (e.g., *prepared-to-commit*) [7]. Components which do not fulfill these far-reaching requirements are excluded from being integrated into the workflow system. On the other hand, workflow systems require a high degree of consistency which can only be achieved by transaction-based systems.

2.3.3 Distribution

Obviously, workflow systems must run in a computer network. Components of the WFMS (kernel and shell) must be accessible from everywhere in the network without regard to their physical location, i.e., the WFMS must offer distribution transparency which in the past lead to the already mentioned client-server architecture of WFMS: controlling, logging, and role binding takes place at the server, actors, tools, and data may be distributed over the network, but all relevant events during workflow execution (e.g., termination of a workflow) must be propagated to the server.

Client-server architectures are only feasible in narrow organizations. In large and distributed organizations, as we envision by our transport company example, WFMS cannot be realized as client-server systems. Also, replication of WFMS at some or all nodes would not help since WFMS server are not stateless and, thus, must closely cooperate.

Distribution is also a fact hard to swallow for WFMS based on active-rule technology. Since there is no context for rules events must be made globally visible to check whether there is some rule in the rule base which is affected by the event. This leads to tremendous communication costs in distributed environments. Newer approaches try to solve this problems by evaluating events in two steps [5]: first, all events which occur at one site are checked by a local event handler and only a predefined subset of events is propagated to a global event handler. By this, event handling is governed by system structure but not by the application semantics.

2.3.4 Cooperation

Often tasks can only be solved by cooperation between actors or tools. Cooperation here means that in a group of actors and/or tools each actor or tool can take the initiative to communicate with others to solve a problem (in contrast to the usual client-server relationship between components). The cooperation consists of a bundle of communications between instances (actors or tools). Since cooperation is aimed to solve a problem, communication along a cooperation must obey certain rules — a *cooperation protocol*. Technically speaking, the execution of a workflow (or task) cannot be mapped to a procedure (or single control flow) but has to be mapped to interactions between several pro-active and re-active components.

In our example workflow (Figure 2) workflow "planning" is a typical example of a cooperative task. While the coordinator defines the set of partners the partners evaluate the order using local information such as the current amount of orders to execute, traffic situation and so on. The coordinator finally decides who will carry out the order. During the whole process all cooperation partners might take the initiative to communicate with each other. For instance the coordinator might cancel the evaluation process due to some unforeseen event or a partner might request further relevant details concerning the freight.

Usually cooperation in WFMS is implemented by sharing information between components, e.g. by input- and output streams as in *FlowMark* [31] or by accessing common databases [45], but explicit communication between components is not supported. We refer to the first as *cooperation by shared information* in contrast to *cooperation by message passing* in the sequel. Cooperation by shared information is natural in client-server environments where control flows and access to data underlies centralized control. In distributed and cooperative environments where multiple control flows may concurrently be active without any centralized control and may need to exchange information with each other without any restriction as to who initiates an information exchange, the message passing paradigm seems more adequate to implement cooperation.

2.3.5 Flexibility

Applications as, e.g., the processing of an order in a transportation company are very complex. Thus, it is normally not feasible to enumerate a-priori all possible workflow executions. Instead one would prefer to describe workflows in an abstract manner. The concrete execution of a workflow will dynamically be

determined at execution time. Unfortunately, events may occur at execution time which have not been considered in the abstract description of the workflow. There are many sources of such events: autonomy of objects (which also includes their potential misbehavior), distribution (e.g., unreliable communication systems), heterogeneity etc.

Take for example our order processing workflow. Congested highways, accidents may disturb the execution of a client's order. Execution of sub-workflows may last an unexpected long time due to the current load of the executing actor (i.e., length of its to-do list). Since components are autonomous they may act in an unexpected manner. For instance, a partner may not evaluate an order although it has been asked to do so, or rejects the execution of an order assigned to it although it accepted it during evaluation. In addition, physical distribution of components can add uncertainty. For instance, a partner site may not be reachable.

Consequently, execution control of workflows must be able to react spontaneously on exceptional events. This flexibility requirement is a further argument to favor declarative workflow models.

3 Object-based Implementation of Workflow Systems

3.1 Mapping Resources to Objects

Object technology appears as a natural technology to meet the requirements of autonomy, distribution, and cooperation because objects are a natural model of interactive components that act in a distributed environment. Consequently, a major premise underlying this paper is that object-systems are a vehicle particularly well-suited to the implementation of workflows. Figure 1 reflects the premise.

We map all resources of the workflow model alike actors, tools, and data to objects. This allows us to treat them interchangeably if this is required in an application context. For instance, actors and tools might be "data" for some applications as, e.g., in an integrated production information system where both planning and production control systems are used together in an integrated environment. In the planning context machines, personnel etc. are subject to planning, i.e., they are "data". In the production control context they take the role of actors and/or tools.

There are differences, of course. Actors and tools differ in their active behavior — as we have already stated above. The first are proactive, the latter merely reactive. However, taking, e.g., the object notion of *CORBA* this seems more an issue of how objects are implemented and less an issue of the expressiveness of an object model. For instance, as a rule of thumb we expect that actors will often be implemented as client objects and tools as server objects. But the distinction between actors and tools is more important at the workflow level than at the object level.

Under these assumptions the WFMS architecture of Figure 3 is revised the architecture of Figure 4. We replace the data manager and tools manager by an object manager. Objects replace both actors and tools.

3.2 Shortcomings of a Pure Object-Based Implementation of a WFMS

Mapping objects to resources let us now face the question how to model workflows. At the leaf-level of workflow hierarchies we use object procedures (or methods) to implement activities. Methods are a natural model for elementary computational units. In some cases procedures may be also an alternative implementation for simple tools. But where would we find complex non-leaf workflows in Figure 4? Clearly, the controller is responsible for their execution — or in the declarative case which we consider our first choice for the workflow level — for guaranteeing the specified interdependencies between activities.

Conceptually one can think of workflows as contexts for the cooperation of those objects that implement resources of a workflow. The cooperation between objects must follow task-specific rules that go beyond the rules which govern the behavior of an individual object and which guarantee the dependencies between activities on individual objects. Thus, dependencies between workflows have to be mapped onto cooperation protocols where dependencies can be described by temporal ordering conditions of messages between objects. The question which now arises is how these cooperation protocols can be best implemented in a distributed object system.

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

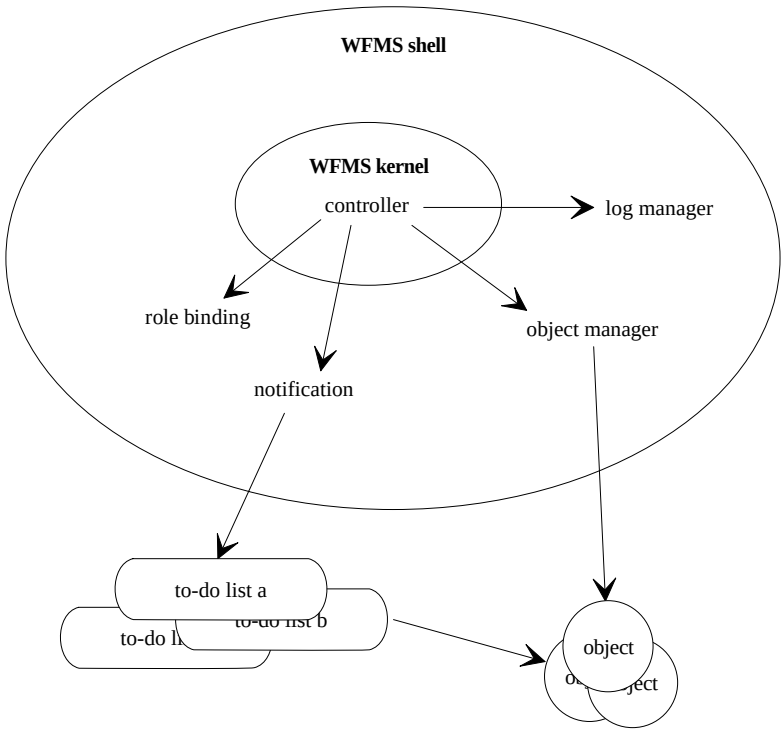


Figure 4: Architecture of an object-based WFMS

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. In complex environments like our transportation scenario such misbehavior may not always 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 [3, 4, 22, 33, 34, 52]. 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 the uncertainties inherent in applications that are part of a large information system. Consequently, we introduce an extended construct which we call an *alliance*.

4 Alliances as a Model of Cooperation in Distributed Object Systems

4.1 Cooperation in Object Systems

We start this section with a brief review of work which in some way relates to our approach.

It has long been recognized that communication abstraction is necessary in object-oriented models. One class of approaches extends interface descriptions of objects by protocols (e.g., [39, 50, 15, 30, 28]) or by a declarative description of object behavior (e.g., by using finite state machines as in [52]). In some cases the separation of interface and implementation was completely abandoned (as e.g., in [53]). 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 [11, 13, 18, 20], 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 [21]. 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 [35]. 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., [40, 51]), 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 [37] 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* [4], 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* [22, 23], *synchronizers* [3], and *adaptors* [52]. 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 participating objects, i.e. the methods 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. [3] mentions distribution but considers it strictly an implementation issue to be solved, e.g., by RPC-style calls.

4.2 Alliances as Materialized Cooperation Protocols

Very crudely speaking, and following the terminology of the ISO/OSI reference model [24], one may view an alliance as an "intelligent" communication channel between two objects, which must be established between them before they can communicate. In fact, however, a more powerful construct is needed.

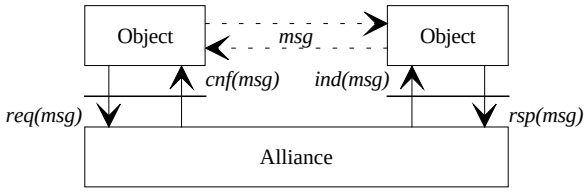


Figure 5: Alliances as communication media between objects

Hence, alliances allow multi-object cooperation where all objects may have the same rights (in contrast to client-server models), may have a life-time that exceeds the life-time or connection time of a connected object, and support a wealth of semantically rich messages. For the objects we assume that they have (at least) one own thread-of-control and, consequently, perform their computations concurrently.

Now suppose that our objective is to express and control multi-object constraints on the messages exchanged between the participating objects. These constraints are always defined in the context of a particular task the objects cooperate on. Therefore, by initiating an alliance with the onset of executing a task, the constraints are being established. The control of the constraints makes use of the communication metaphor as seen in Figure 5: 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 alliance raises the second with the receiver object. Often messages are expected to be answered. In this case we speak of *acknowledged messages*. Two further events will take place when the receiver of a message answers: a message response (*rsp(msg)*) and a message confirmation (*cnf(msg)*). Consequently, no message exchange escapes the attention (and, thus, control) by the alliance.

From a purely structural viewpoint, alliances can be viewed as (dynamic) relationships of a conceptual information system model. Figure 6 shows an excerpt of such a conceptual model of our transportation company using an *OMT*-like notation [44]. According to our discussion of Section 3 resources such as sites, trucks, order, and client are mapped to object types. Likewise, alliances with similar properties are classified into types. Alliance types are denoted as relationship types (diamonds in Figure 6).

Alliances and objects are connected by *roles* (e.g., *coordinator*, *partner* etc. which are denoted as associations between object types and alliance types in Figure 6). We map roles of the workflow level directly onto roles of the object level. On both levels roles define conditions that must always hold for all objects that play this role.

In summary, then, alliances combine static aspects (in the form of relationships with roles) and dynamic aspects (in the form of a communication channel with communication events). We reflect the two aspects in separate parts of the specification of an alliance.

Take the static part. It consists of requirements concerning the message interface of objects. It defines which messages an object can (at least) receive and which messages it can (at most) send. Thus, these conditions constrain the types of objects that can be bound to a role. Figure 7 shows the role definition of the alliance type planning taken from our transportation scenario (**roles** clause).

Each role definition consists of two sets of message type declarations and a role name. Take, e.g., role coordinator. An object which plays this role need not be able to receive any messages and will send no messages other than of type evaluate(...) and assign. While assign is a one-way message (i.e., no answer is expected) evaluate is an acknowledged message with Votes(...) as reply message[5].

Role Partner is *set-valued*, i.e., an arbitrary number of objects can play this role simultaneously. All members of a set-valued role must be type-compatible to the role specification, e.g., in the case of role Partner they must understand messages of type evaluate and must not send any messages. The message type evaluate of role Partner defines again acknowledged messages but this time the message is not answered by a separate message (or call-back) but by an unnamed reply[6]. Unnamed replies allow objects to take part in an alliance and to answer messages without any knowledge about this alliance and interfaces of their cooperation partners. This is necessary to build pure server objects (or tools) independently from possible collaborations.

[5] A reply message can be compared with a call-back function.
[6] Nierstrasz introduces *private channels* for this purpose [39].

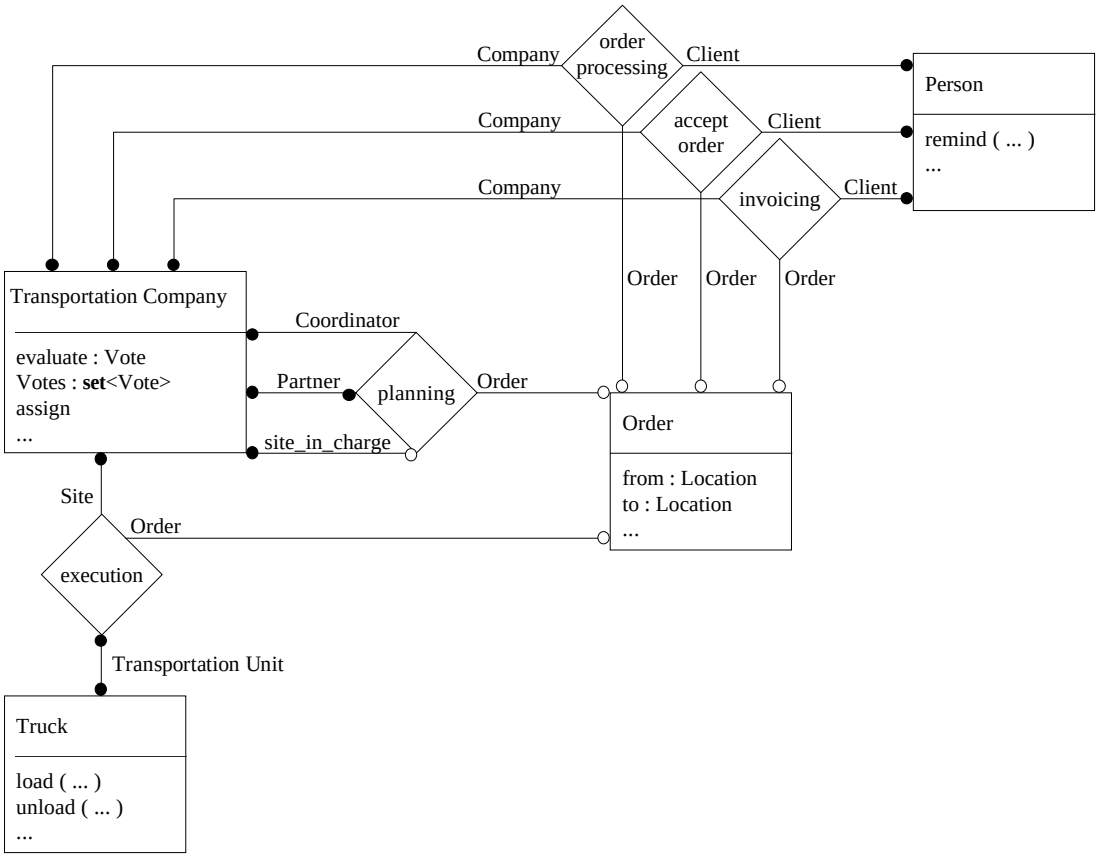


Figure 6: Conceptual schema of a transportation company

The dynamic part of an alliance is subject of the next section.

4.3 Rule-based Specification of Alliances

The dynamic part of an alliance takes its cues from telecommunications where the interrelationships between and constraints on events are specified in the form of a protocol. The protocol specification of an alliance consists of three parts: a definition of a set of possible states, an initialization, and a set of communication rules that define upon which events (*req* and *rsp*, cf. Figure 5) the alliance changes its state, what state is reached, which events the alliance raises with objects, and how the set of participants evolves. Rules are a proven technique for protocol specification in telecommunications [46].

4.3.1 An Example Protocol

In order to illustrate our example protocol we first give an abstract specification of this protocol as an automaton. We use an OMT-like notation to describe the dynamics of alliance type planning (Figure 8). When the coordinator sends an evaluation message to its partners (*eval?*) the alliance reaches the state *eval in progress*. As long as partners answer to this message (*vote!*) the alliance remains in this state. When all partners have voted condition (*[all eval]*) is satisfied and the state *eval complete* is reached. Now the coordinator can repeat the evaluation process which would cause the alliance to return to the state *eval in progress*, or it can assign a partner to execute the order (*assign!*) which terminates the alliance. In each state the coordinator may finish the evaluation process and direct a partner to execute the order (*assign!*), e.g., because there is no time left for planning.

```
alliance planning {
roles:
    {   // can receive
        // nothing
    }
    {   // can send
        evaluate(set<Object>) → Votes (set<Vote>);
        assign (Object);
    }   coordinator;
    {   // can receive
        evaluate() → evaluate()::reply(Vote);
    }
    {   // can send
        // nothing
    }   set<Partner>;
    {   // can receive
        assign();
    }   site_in_charge;
    {...}{...} Order;
    {   // can receive
        timeout(int);
    }
    {   //can send
        alarm(int);
    }   Timer;
...
```

Figure 7: Alliance type planning: roles

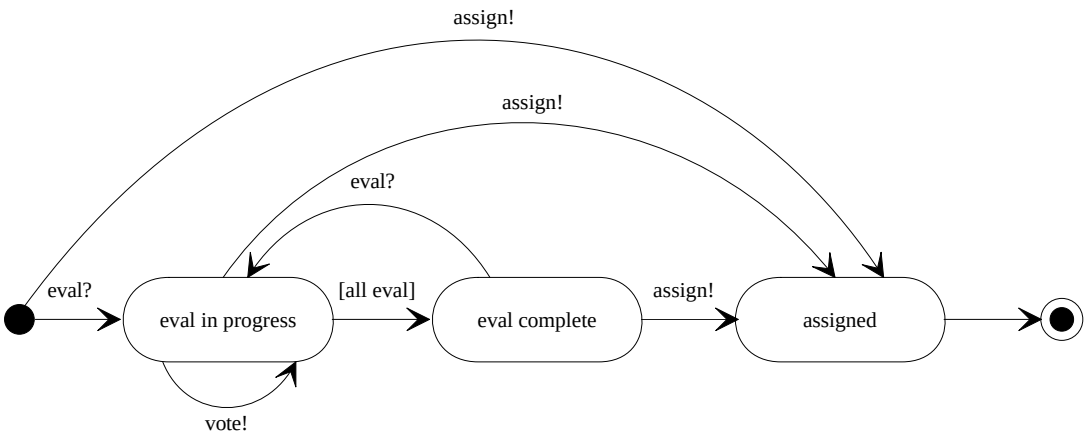


Figure 8: Example protocol specification

4.3.2 States

We implement this protocol using the alliance construct. We first have to define the set of possible states. This is done by a set of typed variables as can be seen from Figure 9.

The boolean variable eval_in_progress models the states *eval in progress* and *eval complete*. We do not need a third value to represent *assignment* because this state is redundant with the final state, and has only been introduced above to clarify the semantics of the protocol. The second variable votes is used to refine the definition of state *eval in progress*. It contains the votes for the orders as returned by the

```
alliance planning {
roles:  ...
states:
    bool eval_in_progress;
    set<Vote> votes;
...
```

Figure 9: Alliance type planning: states

partners.

4.3.3 Initialization

```
alliance planning {
roles:
states:
initialization:
    planning (Object k, Object a, set<Object> p_set) {
        Coordinator = k;
        Order = a;
        Partner.insert(p_set);
        Timer = new Clock;
        eval_in_progress = false;
        votes.empty();
        persistent;
    }
...
```

Figure 10: Alliance type planning: initialization

When an alliance is instantiated it has to be initialized. The initial state is defined by assigning values to the state variables. Additionally, roles may initially be bound to objects. Figure 10 shows the initializion part of our example alliance. At instantiation time the coordinator, the order to plan, and an initial set of partners are determined. We use simple assignment operators and a predefined insert operation on set-valued roles to denote this. But as we will see in Section 7 role binding cannot be done by simply writing a value to a memory location but is a quite complex operation. Besides binding objects to roles that have been given as parameters of the initialization operation or — as we will see later — of messages, an alliance can create objects on its own as can be seen from Figure 10 where an new object of a predefined type Clock is created and bound to role Timer. We will later use this object to implement exception handling by a timeout mechanism. 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 Section 7.

4.3.4 Communication Rules

The communication rules of our example alliance are given in Figure 11. Communication rules map message requests guarded by an optional condition to a reaction. The definition of a message request consists of a message (perhaps parameterized) and a role name (following '@') which denotes the originator of the message request.

The condition is given by the expression following **if**. A communication rule can only "fire", i.e., the specified reaction (the code between { }) is executed, if the given condition evaluates to true. Thus,

```
        alliance planning {
        roles: ... states: ... initialization: ...
        rules:
(1)         evaluate(p_set)@coordinator if not eval_in_progress {
                Partner.insert(p_set);
                timeout(TIMEOUT)@Timer; evaluate()@Partner*;
                eval_in_progress = true;
            }
(2)         evaluate(p_set)@coordinator if eval_in_progress {
                // ignore multiple evaluation requests
            }
(3)         evaluate()::reply(w)@Partner if eval_in_progress {
                votes.insert(w);
            }
(4)         if votes.size == partner.size {
                coordinator.Votes(votes);
                votes = set<Vote>::empty; sites = set<Object>::empty;
                eval_in_progress = false;
            }
(5)         assign(f)@coordinator {
                assign()@site_in_charge;
                site_in_charge = f; Timer.delete();
                terminate();
            }
(6)         alarm()@Timer if eval_in_progress {
                coordinator.Votes(votes); // only partial votes are delivered
                votes = set<Vote>::empty; sites = set<Object>::empty;
                eval_in_progress = false; Timer.delete();
            }
        };
```

Figure 11: Alliance type planning: protocol rules

the code of the first rule in Figure 11 is only executed when the specified message request of the coordinator has occurred and the variable eval_in_progress evaluates to false. Conditions are restricted to a boolean expression over state variables and message parameters of the request, and must not contain any interaction with objects.

On detection of a message request or on reaching a certain state (as, e.g., in rule (4) of Figure 11 where size is a predefined operation on set-valued roles and state variables which returns the number of elements) an alliance may react by modifying some local state variables (e.g., in the first rule the variable eval_in_progress is set) and/or by modifying some role bindings (again the first rule is an example: the coordinator can extend the set of partners he wishes to cooperate with), and/or by indicating messages at roles (e.g., an evaluate-message is indicated at role Partner in the first rule), and/or by terminating (e.g., rule (6) will lead to termination). Termination means that the alliance will not handle any further message requests (the "connection is closed").

In the case of set-valued roles messages can either be indicated with all objects bound to that role which is denoted by '*' (as, e.g., in the first rule), or to one arbitrary member selected indeterministically by the system. The first option is useful to implement queries to object sets. For instance, we can view the evaluation of an order by a set of partners as a query from the coordinator to a set of partner objects. The latter option is useful if a set of objects offers equivalent services and it does not matter which individual object does actually perform the service, but where redundant objects may help to enhance the overall reliability of the system.

In order to deal with message requests from set-valued roles we allow to refer to the originator of the message by binding it as a special event parameter. For example, one might be interested in the sender of each vote in the planning alliance, which can be specified as follows:

```
evaluate()::reply(w)@Partner p if ⟨some condition over p⟩ {
    ⟨do something with p⟩
}
```

Set-valued roles play an important role because they ease the task to implement set-oriented computations in distributed environments and may be of particular benefit to object systems containing large sets of objects.

Alliances can control communication between objects in many ways. They can simply transport messages from a sender to a receiver. They can ignore unexpected messages in order to protect objects from illegal invocations. The second rule of Figure 11 gives an example: once the evaluation has been started no further evaluation-messages from the coordinator are delivered. They can accumulate messages using their internal state, and indicate at an object just an aggregation of all messages. Rules (3) and (4) of Figure 11 use this mechanism to collect all votes of the partners and indicate them as one message with the coordinator (Votes in rule (4)). The evaluation is complete if the set of votes contain the same number of elements as the set of partners. But we cannot be sure that this state is ever reached. By using special clock objects alliances can realize timeout mechanisms which put alliances into a position to cope with a situation in which expected messages do not arrive. We used this in our example to guarantee an upper time limit for the evaluation process. When the coordinator requests an evaluation (first rule in Figure 11) a timeout is defined by indicating an appropriate message at the Timer. When a timeout occurs (rule (6) in Figure 11) the partial result of the evaluation is delivered to the coordinator. This is an example of a fault-tolerant "query protocol" which might often be applicable in the case of large sets of queried objects in a distributed environment where it is a better solution to deliver partial results of a complex query than nothing at all.

As a consequence of these sophisticated communication control mechanisms alliances can be used to bridge type incompatibilities between cooperating objects and, hence, offer the functionality of *adaptors* [52]. *Synchronizers* [3] can also easily be realized using alliances. Beyond this alliances can provide their participants with guarantees with respect to temporal ordering of messages (or method invocations). As a consequence of our assumption that objects are autonomous entities, alliances cannot prevent objects from behaving erroneously but they can protect other participants against faults. In a nutshell we accept the fact that in a complex and distributed system there will be always erroneous or inconsistent objects, but we try to prevent them from doing any harm within the object system.

4.4 Semantics and Execution Model

The semantics of alliance specifications can be defined by labeled transition systems (LTS) [6]. A LTS consists of a set of states and a set of transitions. The transitions are labeled. The set of states of an alliance is derived from the state variables. The role specifications define the set of labels. We skip the details of mapping alliance specifications to LTS because they not relevant in the context of this paper. The transition system semantics of alliances implies that alliances work sequentially. Since objects act concurrently and, thus, may request messages simultaneously, message requests must be ordered in a system-defined way. This ordering is performed by a so-called *evaluation cycle* through which an alliance loops from instantiation until termination and which defines the execution model of alliances. The evaluation cycle works like follows:

I. Select a role indeterministically and fairly. Set-valued roles are treated as a set of single-valued roles.

II. If there is a message request with the selected role execute the following steps:

   (a) Compute the set of rules which wait for this message request and for which their conditions evaluate to true (i.e., the rules that can "fire").

   (b) If more than one rule qualifies select one indeterministically.

   (c) If no rule qualifies for execution go to step I.

   (d) Execute the specified reaction by evaluating the specified effects in the following order:

      1. role bindings
      2. indications

3. state transitions

(e) Discard the message and go to step I

Evaluation of a message request, i.e. step II, is executed atomically. The transition system semantics of alliances implies atomicity of rule execution.

Rules without any message request, so-called *immediate rules* as, e.g., rule (4) in Figure 11 conceptually fire on the occurrence of an event raised at a so-called *anonymous role*. This role can be selected as the result of step I just like any other role of the alliance. Thus, every immediate rule

```
condition { }
```

can be translated into

```
local_event@anonymous_role if condition { }
```

where always a local_event has occurred when anonymous_role is selected for evaluation.

4.5 Alliance Hierarchies

As Figure 2 indicates the planning of an order is a sub-activity of overall processing of an order. One should be able to model hierarchical structures of activities with alliance as well. In order to meet this requirement we have to add two minor extensions to the alliance model: First, we allow that alliances can create sub-alliances. Second, we must notify an alliance if one of its sub-alliances terminates. Sub-alliances must be registered as part of the state of their father. The evaluation-cycle of sub-alliances is executed concurrently to that of their father, i.e., sub-alliances are executed independently from their father.

```
alliance order_processing {
roles:
    {...}{...} Company;
    {...}{...} Client;
    {...}{...} Order;
states:
    enum(acceptance, planning, execution, invoicing) current;
    anAlliance sub;
initialization:
    order_processing(Object c, Object s, Object o) {
        Client = c;
        Company = s;
        Order = o;
        current = acceptance;
        sub = new acceptance(Client, Site, Order);
    }
rules:
    ...
    terminate(sub) {
        current = planning;
        sub = new planning(Site, Order);
    }
    ...
```

Figure 12: Alliance type order processing with sub-alliances

Figure 12 shows a part of the alliance type order_processing. At instantiation time a new sub-alliance of type acceptance is created. On its termination a second sub-alliance, now of type planning, is created.

5 Mapping Workflows to Alliances

In this section we will illustrate by examples how elements of a execution control specification of workflow models can be mapped onto alliances. We discuss the implementation of procedure-like (i.e. imperative) and declarative control structures.

In the sequel we denote by i, i1, i2, ... arbitrary events (requests, local events) that may lead to the execution of a rule. a, a1, a2, ... denote either message indications or creation of new alliances. c, c1, c2, ... are response or termination events with c matching a and ci matching ai. Finally b, b1, b2, ... are boolean variables.

5.1 Mapping Imperative Execution Control

The elementary building blocks of procedure-like (imperative) execution control descriptions are sequential execution, conditional execution, loops, and parallel execution (Section 2.1.3). Sequentialization of activities ($a_1; a_2$) can be enforced by chaining rules as shown in the following example (we assume that b is initially false and will not be set to true by any other rule):

```
i {a1; b = true}
b {a2}
```

Note that due to the execution model for alliances the rules only enforce that a1 takes place before a2 and that no additional requests from objects are necessary to let a2 take place. Hence, the alliance is indeed the driving force to let a2 take place after a1. On the other hand, alliances add a certain degree of ambiguity. They do not preclude that other events may occur between the execution of a1 and a2, or that other rules may fire in between. If, true to strict sequential execution, this is to be prevented all other rules of the alliance type must be masked with **not** b, i.e. must have the form ...**if not** b { }. In general rule-based protocol specification allows for much more varieties of "sequential execution semantics" than can be expressed by a script-like specification.

Conditional execution of activities ($b?a_1 : a_2$) can be realized by alliances with the following rules,

```
i if b {a1}
i if not b {a2}
```

under the further assumption that no other rules can fire on the occurrence of i.

An easy implementation of loops (**while** $b : a$) might look as follows:

```
i if b {a; b = ...}
b {a; b = ...}
```

Of course, there may be rules that fire in an interleaved fashion with our "loop rules". This has two consequences: First, the loop variable b might be affected by other "non-loop rules" which might either lead to an unexpected termination of the "loop" or to infinite "loop execution". Second, activities (indication and instantiation of new alliances) may be initiated between two executions of the loop body (i.e., the action part of both loops). To preclude interleaving, i.e. to obtain the traditional atomic behavior of loops, requires additional linguistic effort, for example by masking all other rules with **not** b, i.e. each other rule must have the form ...**if not** b {...}.

Realization of parallel execution of activities ($\|a1, a2, \ldots$) is trivial due to our assumption that objects perform their operations concurrently (cf. Section 4.2):

```
i {a1; a2; ...}
```

5.2 Mapping Declarative Execution Control

For the elementary building blocks see again Section 2.1.3. Temporal ordering conditions ($a_1 < a_2$) can be implemented by alliances as follows (we assume that b1 has initially been set to false):

```
i1 if not b1 {a2; b2 = true}
i2 if not b2 {a1; b1 = true}
c1 {b1 = false}
```

In order to guarantee that a1 does not take place once a2 took place each rule that contains a1 in its action part must be masked with **not** b2. The third rule ensures that a2 does not take place until termination of a1 (c1 is either a response or a termination event). The rules realize a special interpretation of the temporal ordering condition, i.e, that a response to a1 or a termination event must have occurred (i.e., the activity represented by a1 must have been completed) before a2 can start. One can realize this interpretation only if the first activity is represented by an acknowledged message.

Existence conditions ($a_1 \Rightarrow a_2$) can be translated to rules as follows (we assume that b initially is false):

```
i1 {a1}
c2 {b = true}
i2 if b {terminate}
```

Once a1 took place the variable b ensures that the alliance does not terminate before a2 terminates (c2) if we assume that all rules containing termination in their action part are masked by b, i.e., have the form ...**if** b {...; **terminate**} and no other rules than those reacting on c2 set b to true. There must be one or more rules which let a2 take place in their action parts. If the existence condition had a slightly different semantics, e.g., requiring only that a2 has been started but does not depend on its termination, rules which have a2 in their action part must also set b to true.

6 Integration of Alliances into Distributed Object Systems

If we wish to integrate alliances into distributed object management systems this should be done on the basis of one of numerous existing approaches of distributed object management. In this section we discuss how this can be done. We choose *OMG's CORBA* [41] as an example.

6.1 Distributed Object Management

Distributed object management software is a system that allows the storage, activation, and communication of objects in a computer network. An object consists of a system-wide unique logical, i.e., state and location independent, identifier (OID), a set of message types (services), a hidden state, and a hidden implementation of the services. Objects are units of distribution, i.e., they are neither distributed across more than one process nor across more than one database.

In a distributed object system an object can invoke an operation (send a message) at another object at a different process or node of the network — a *remote object* — almost as easily as it can invoke an operation on an object within the same process.

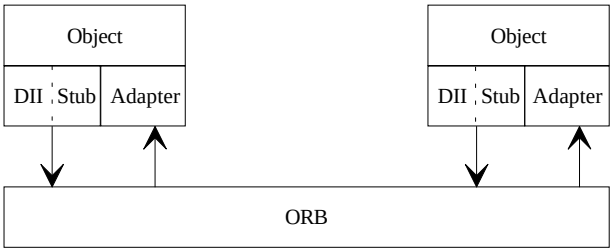


Figure 13: Distributed Objects in *CORBA*

If an object contains a reference to a remote object, requests to this object are redirected to a so-called *request broker* which localizes the referenced object, "activates" it if necessary, i.e., assigns a process to it[7], and indicates the request. *CORBA* offers two common techniques to achieve this: a stub and a dynamic invocation interface (DII) to the request broker (ORB) (cf. Figure 13). The first allows for static type checking of remote references and invocations. A stub — also often called a *proxy* [47] — is a

[7] A set of objects can share a single process to save resources (cf. Section 8).

local representative of a remote object and is created in the client's process when the client assigns the identifier of a remote object to a reference variable. In the case of dynamic invocation requests are objects which a client creates when he invokes a remote object. The necessary type information is provided as run-time parameters. Dynamic invocation is useful when type information for remote objects cannot be provided statically, or when script languages are used to implement objects. The ORB uses an *adapter* to deliver requests to an object.

Method invocation on remote objects can either be synchronous, i.e., the client is blocked until the method is executed and the server returns a result, or asynchronous, i.e., non-blocking. Asynchronous communication is important in cooperative environments since, as we have already outlined above, in most cases activities are too complex to use procedures as an adequate abstraction. Also, cyclic communication structures (e.g., A requests B to perform a service S, subsequently B asks A for further information in order to perform S) are quite common. Here, asynchronous communication is required to avoid activities to become "deadlocked".

6.2 Integration of Alliances

In order to integrate alliances into *CORBA*-like distributed object systems there are two options: Either alliances are implemented as "first-class" objects (cf. Figure 14), or their implementation is distributed pretty much like the distributed implementation of a layer in a protocol stack (cf. Figure 15). In the first case alliances could be compared with some kind of *adapter*- and *mediator*-objects as identified in [16] and extended to a distributed and concurrent environment. Our contribution would then be more in the direction of a design methodology for distributed applications than a technical innovation.

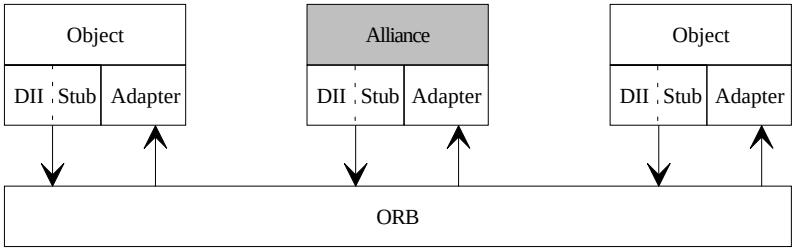


Figure 14: Alliance as "first-class" objects in *CORBA*

The main disadvantage of this solution is that the objects must use communication subsystem to interact with alliances which leaves it to the objects to add higher-level communication functionalities to deal with its hazards and uncertainties (Figure 14). These are more naturally to hide if the second, the distributed implementation variant, is used (Figure 15).

In this variant each participant object of an alliance references its own local representative of the alliance. Alliances become generalized smart proxies: Generalized in the sense that they connect a collection of objects and are not restricted to two objects; smart because they realize a context-sensitive cooperation protocol, i.e., contain their own state information and program code. Figure 15 illustrates how this idea can be extended into the *CORBA* architecture. Each participant of an alliance interacts with a local representative of this alliance (*Alliance rep.* in Figure 15). The representatives of an alliance communicate with each other by using standard communication services.

The interface between objects and alliances can be statically or dynamically typed. In the first case we must extend the object language by constructs to mark reference variables of object types as references to alliances. A possible *ODMG*-like [10] syntax is shown in Figure 16. In this case conformance to role specifications can be checked statically when objects are compiled. The keyword **as** which we used in Figurefig:Object:Interface can be interpreted as a generalized **inverse**-clause as proposed in the *ODMG* standard to model binary relationships (also known as inverse references). In this case it does not point to an attribute of the object which is referenced but to a role name of an n-ary dynamic relationship.

Note, that it is not always necessary to declare references to alliances explicitly. In the case that objects are mere "servers" in an alliance, i.e., do not take the initiative for any communication on their

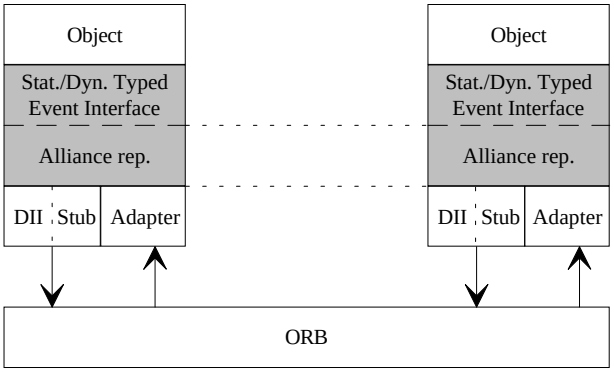


Figure 15: Alliances as "protocol layer" in *CORBA*

```
interface Transportation_Company {
    ...
    partner: set<planning> as planning::coordinator;
        // I'm the coordinator and this is the "pointer" to my partners
    coordinator: set<planning> as planning::partner;
        // I'm a partner and this is the "pointer" to my coordinator
    ...
};
```

Figure 16: Statically typed event interface

own (formally: do not send any other messages besides unnamed replies) no reference to an alliance must be declared. Consequently, the variable coordinator in Figure 13 could be left out.

In the dynamic case objects hold untyped references to alliances. Requests are issued by creating new request objects which take the requestor's role name and the requested message (name and message parameters) as parameters. Conformance to role specifications must be checked at run-time.

7 Distributed Implementation of Alliances

In this section we have a closer look at the distributed implementation of alliances as proposed in the last section. We must considerate two major issues: first, how objects are associated to alliances (role binding) and, second, how the evaluation cycle of an alliance can be implemented in a distributed environment. This issue is closely related to the question how and where to maintain the state of an alliance.

7.1 Role Binding

Binding an object to a role is the process of establishing a typed bi-directional association between an object and an alliance. Role binding takes place as part of the execution of some rule or during initialization (see, e.g., Figures 11 and 10, respectively). It covers the following steps:

1. Given an OID of an object to be bound, the object's current location must be determined. For this purpose the alliance accesses a global object index.

2. A new representative is created for the object to be bound.

3. In the case of dynamically typed event interfaces the type of the object to be bound can partially be checked. If the interface specification is available the set of receivable message types can be

checked. The set of send-able message cannot be checked statically because objects contain only untyped references to alliances even if the implementation of object types is available. Consequently, message requests that do not meet the role specification can only be rejected when they actually occur.

If the statically typed event interface is used no type errors can occur at run-time.

4. The role is updated with the new object identifier.

5. The object is notified about the new binding in order to let it update its state with a reference to the newly created representative of the alliance.

7.2 Implementation of State and Evaluation Cycle

Objects are located across a network, where each object raises communication events with local representatives of alliances to which it has been bound. Consequently, from a functional perspective an alliance is itself distributed. This leaves considerable latitude to the implementation of alliances because the questions how to implement the evaluation cycle, i.e., the internal control flow of an alliance, and how to implement its state may now be answered in more than one way. Special care must be taken on fairness of role selection and atomicity of rule execution. Since representatives are distributed across several nodes and processes, the following dimensions define a space of implementation variants:

1. Shall alliances be executed at one place or shall their execution be distributed across the set of involved nodes?

2. Shall the state be stored at one place or shall it be replicated[8] at all or a subset of involved nodes? We must further distinguish between the main-memory representation of the state (subsequently called transient state) and its representation on durable storage (persistent state).

We obtain eight implementation variants: control central or distributed, transient state central or replicated, persistent state central or replicated. However, if control is distributed it does not make much sense to keep the transient state at one remote node because we would not gain anything concerning reliability but would have to pay additional communication costs. It also does not sound very clever to replicate the transient state if control is at one node, since only the evaluation cycle needs access to state information. As a general rule we can postulate that transient state information should always be there where control is[9], and access to transient state information should never be remote to reduce communication costs. Consequently, we can immediately exclude four variants.

Figure 17 illustrates the remaining four variants. The upper left figure shows a variant where both state and control are centralized. Fairness can easily be achieved by simply iterating through all roles. Those representatives which are temporally not reachable (e.g., because their nodes are down) can simply be skipped to enhance fault-tolerance. Atomicity has to rely on services of the next lower layer (Figure 1). For instance, execution of a rule could be an atomic transaction where transactions semantics must be available for both updating the state and raising message indications with the (remote) representatives. For example, if one of the nodes to which messages should be delivered is not reachable it must be possible to roll back all updates on the state and all former message indications. In this variant the local representatives are degraded to be mere event interfaces to objects (comparable to stubs). As the figure indicates one representative takes the role of the chief (the shadowed representative), i.e., executes the evaluation cycle and maintains the alliance state locally.

This variant is easy to implement and requires a minimum of additional communication (compared to the variants which we discuss subsequently). On the other hand it is vulnerable — as any centralized architecture — against the failure of the chief or its node since all participants at the alliance will suffer in this case.

The upper right figure shows a fully distributed variant: state is replicated and control is executed cooperatively at all nodes. In this variant control is always at the node where a message request that

[8] We do not consider fragmented states, since it is in general not possible to compute an appropriate fragmentation of the state automatically from a given alliance specification and we do not want to bother a system implementor with further implementation overhead. But it could be worthwhile to investigate fragmentation in future research.

[9] Therefore, we will use the terms state and persistent state interchangeably in the sequel.

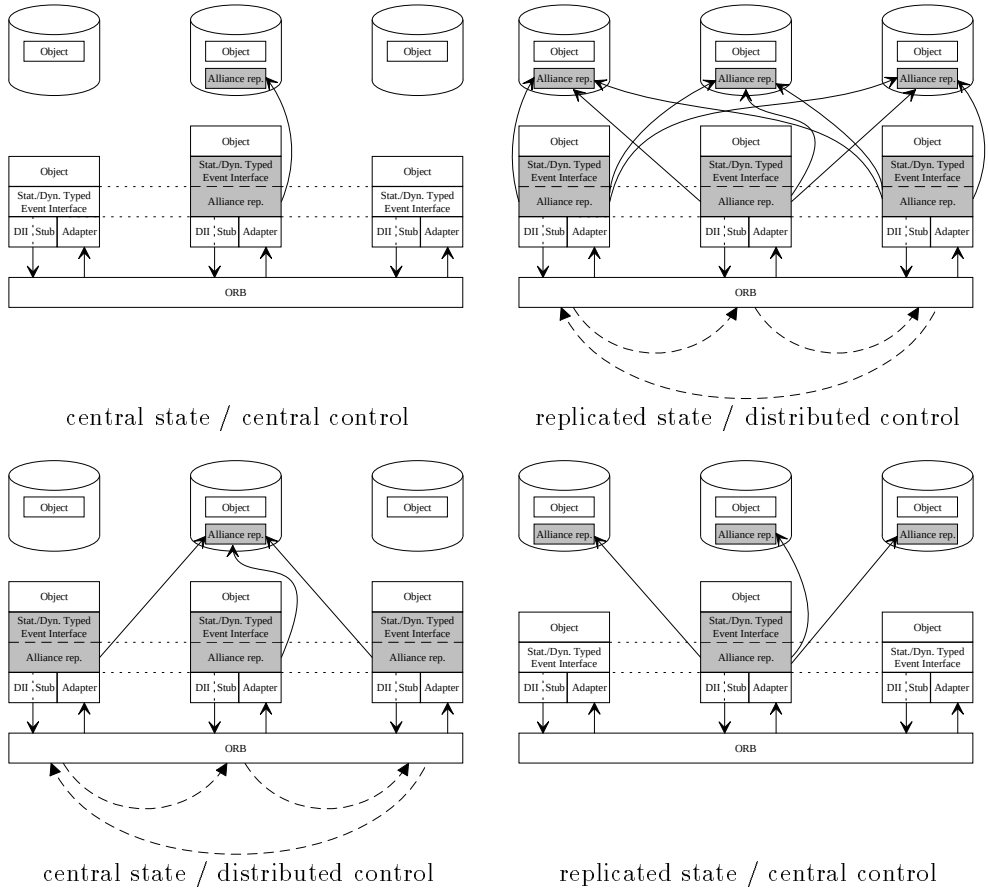


Figure 17: Implementation variants

is to be evaluated has occurred. The fairness of role selection can be guaranteed by choosing a token-ring algorithm which skips non-reachable representatives only a finite number of iterations. If the given network and underlying services do not meet the requirements for reliable token-ring algorithms (e.g., if the net might be partitioned or node failures cannot be detected) we could alternatively regard the set of representatives as active replicates and choose some of the well-known replication algorithms (see, e.g., [26]). Note that in this case it is much harder to formally guarantee fairness of role selection. The flow of control among the representatives is symbolized by dashed arrows in Figure 17.

Atomicity can be achieved in a similar way as in the first variant. But this time we have to apply distributed transactions since all state replicates at several nodes must be atomically updated. In order to guarantee consistency of replicated states well-known database algorithms can be applied (see, e.g., [43]).

Implementation of this variant is quite complex. Furthermore, we have to account for a considerable additional communication overhead. One may wonder whether the disadvantages of distributed implementation of alliances do not outweigh their advantages such as increased reliability in that even if some nodes are temporally not available the cooperation along an alliance will continue. After all quite reliable standard services for communication and databases are available. But not only enhanced fault-tolerance of the WFMS but also organizational restrictions may still require a distributed implementation. Consider, e.g., our truck scenario. There, a manager at one site may wish to have a look at the state of the alliance in order to control the progress of activities. Even if communication costs might not be an argument because fast networks and protocols are available there may be organizational restrictions which necessitate a replicated implementation. For instance, each site of our truck scenario may be an independent company just loosely cooperating with a set of other truck companies. In this case our manager would need direct access to resources of one of his or her partners which will not be granted

because of accounting or security reasons. Consequently, local availability of sharable state information could be a functional requirement in situations where alliances are to be monitored at arbitrary locations of a distributed system.

The third and forth variants (lower left and lower right in Figure 17) are mixes between the first two variants. The variant illustrated in the lower left corner distributes control but maintains state centrally. This prevents the overhead to keep the state consistent but retains the benefits concerning fault-tolerance if we assume that a node failure does not prevent access to state information. The last variant complements the third: it replicates the state but control is centralized. This variant might be taken into consideration if fault-tolerance of alliance execution is not an issue but availability of state information is important.

Whichever alternative is selected, a distributed implementation of alliances hides all inter-process-communication behind the interface of alliances, and allows to handle errors, especially those which can occur in distributed environments, in an application-specific and at least partially transparent manner. Thus, alliances contribute to distribution transparency and add substantial functionality to the object layer of Figure 1.

8 Standard Services and Protocols

Distributed object management and, hence, distributed implementation of alliances heavily rely on standard services such as database, thread, and process services. We distinguish between local and global services. A local service is a service which is only available to one node[10] in a network. Global services must be available to all nodes in a like fashion. A similar distinction can be made for the information, e.g., object types and states, which must be maintained in a WFMS. Some information is only required locally, other information must be globally accessible via global services.

8.1 Local Services

8.1.1 Database Services

We use database services to make objects and alliances persistent which is required since both may be long-living. In addition, database services can be used to define atomic computational units of objects by means of transactions. Databases need not to globally accessible because (a) objects are encapsulated, i.e. the state of an object is only accessed by itself, and (b) the replicated state implementation alternatives for alliances let representatives maintain state information locally and do not require access to shared state information. A replication control algorithm can be used to keep the state consistent. In the case the state of an alliance is centrally maintained we need global database services for this purpose.

Local database services can also be used to maintain type information for objects. As we saw above, type information must be available when an alliance binds an object (referenced by an OID) to a role. Since only the local representative must access the type information (interface and implementation) of the object just a unique type identifier is required globally.

8.1.2 Thread and Process Services

Objects must be assigned to threads and processes. For this we need thread and process services. Since processes are only temporally active, alliances may indicate messages to objects which currently are not assigned to a process but "sleep" in a database. Consequently, a process service must assign an object to a process — either by creating a new process or by using an active one. In *CORBA* the adapter (cf. Figure 15) provides process services [41].

Fine-grained active objects should be implemented using threads as, e.g., offered in *OSF-DCE* [42], such that a set of active objects can share a common process, since processes are a very expensive resource. High-level language primitives which allow a programmer to implement active objects without having to explicitly program the threads, as offered in so-called *concurrent object-oriented programming languages*

[10] A node is a logical unit in a network and does not necessarily refer to a workstation or PC. Depending on the given network architecture and software a node may be a (temporally active) process together with a database, a workstation or PC where a set of processes may maintain their databases by a common database server, or a LAN with a network file system where processes may be run on several processes but use a common database server to maintain their databases.

[2], ease the task of implementing fine-grained active objects. There exist numerous approaches in the literature, e.g., [27, 53, 19, 28].

8.2 Global Services

8.2.1 Distributed Database Services

Two types of information must be globally available: an object index which maps OIDs to their current location, i.e. a node, and type information on alliances.

As already mentioned in Section 7, alliances need access to a global object index in order to localize objects which are to be bound to a role. It is natural to consider distributed database technology to maintain this index. Both objects and alliances must access this index to update it when they create new objects. Localization of objects, i.e. a read access, is only required for alliances when they bind objects to roles. In order to avoid too many accesses to the possibly very large index, addresses can be materialized inside the alliance, although this leads to the problem of potentially invalid addresses if objects can change their location. As an ad-hoc solution for this problem the entry of a newly bound object in the index could be extended by the identifier of the alliance in order to be able to update the address stored in the alliance when the object is moved. Alternatively a common proxy technique from distributed systems can be used [47]. Future research should pay further attention to mobile objects because they seem to become a very important feature for distributed applications.

Access to the object index must be synchronized by transactions since more than one object or alliance may use it concurrently. Note that the object index need not contain the exact address of an object but only its current node. Thus, its maintenance can be decoupled from the maintenance of local object tables used by local database services (see above).

Alliance type information must be globally available since every object at every node may create a new instance of a certain alliance type.

The atomicity of step II of the evaluation cycle (Section 4.4) can be achieved by embedding it into a transaction. Since alliance states may be replicated a distributed commit protocol (e.g. 2PC) must be used. A distributed commit protocol meets also our requirement for atomic propagation of message indications to remote representatives.

8.2.2 Communication Services

It is obvious that communication services as, e.g., offered by the *CORBA* standard play a key role in the implementation of WFMS. Alliance representatives use synchronous and asynchronous RPC to communicate with each other. Multicast — though not yet included in *CORBA* — would be beneficial to support indications at set-valued roles. Note, however, that objects obtain high-level communication services via alliances rather than the lower-level primitives still enforced by distributed platforms.

9 Discussion

First we briefly review how we implemented the components of a WFMS as identified in Figure 4 by using alliances.

The set of alliances which exist in the system, and their evaluation cycles implement the controller of the WFMS kernel in a distributed and decentralized fashion. Local databases and the global object index implement the object manager also in a distributed way. Alliances realize notification by indicating messages with objects and notifying them about role bindings. The implementation of role binding has been discussed in Section 7. Of course, a workflow model should also offer more abstract and declarative ways to specify actors which qualify as participants at a workflow rather than enumerate them individually by their name. Thus, Section 7 assumes that a mapping from a declarative specification of actors to OIDs has already been done. We left out logging, the last component of the WFMS shell, in this paper.

Subsequently we discuss how the proposed implementation of WFMS meets the requirements listed in Section 2.3.

Scalability and Extensibility mainly depend on the number of and access rates to centralized components and global data structures in the system. The most important issue here is event handling.

Notification and controller of Figure 4 are responsible for event handling. In a centralized solution their performance would depend on the number of objects and workflows which raise events.

In an alliance-based implementation event handling is independent of the number of active workflows because events are only visible in the context of a single alliance. The number of participants especially if we consider set-valued roles with potentially large numbers of members can influence the throughput of a workflow. But this is restricted to this particular alliance and leaves parallel active alliances unaffected.

Furthermore, an alliance-based system does not need any central server component. Every object is its own server and uses only local services. Alliances uses local services and global communication services.

An alliance-based implementation needs two kinds of global data structures: type information of alliances (rules etc.) and an object index. If we assume that modifications on type information is quite rare we can fully replicate all alliance types at all nodes. If objects use the statically typed event interface run-time accesses to type information can be reduced.

The object index is more critical with respect to scalability, since this index must be accessed quite frequently. Read accesses to the object index can be reduced if object addresses are kept with the roles in alliances as already sketched in Section 8.2.1. But this does not influence the overhead for maintaining the index when objects are created or deleted. Some relaxation can be expected if the index is implemented as, e.g., proposed in [32]. If objects are immobile the object index can be completely abandoned because physical object identifiers can be used. Note, that this problem is well-known in distributed database technology [43] and has not been caused by the introduction of alliances. It should also be clear from the former sections that there is no need for a global alliance index, i.e., alliances do not aggravate the problem.

Integration of Autonomous Components How easy or hard it is to integrate an autonomous component into a system depends on the requirements which the component has to meet in order to be integrated. The more one expects from the behavior of a component the lower is the chance that some given component can be integrated. WFMS which are based on distributed transaction systems make far-reaching assumptions about objects they want to integrate [7]: objects must offer certain messages at their interface (e.g., *commit* and *abort*) and they must reach certain internal states (e.g., *prepared-to-commit*). Components which do not meet these requirements are excluded from integration.

In contrast alliances expect very little from their participants. No special messages are required for the objects' interfaces. Objects must just behave consistent with some "law of nature", i.e., elementary causalities. This allows the integration of almost any objects.

In addition, integration is supported by the capability of alliances to bridge type and interface protocol[11] incompatibilities between cooperating objects.

Integration of legacy software has not explicitly be mentioned here. Usually legacy software is "wrapped" to give it the odium of an object. This wrapper might contain references to alliances in order to let a legacy object take part in a cooperation. This technique is mainly applicable to integrate mere server objects. If legacy software should take the initiative to communicate in a cooperation, the implementation of the wrapper will grow quite complex since communications initiated by the software must be "trapped", i.e., caught and redirected to an alliance. In some cases even a reimplementation might be necessary.

Distribution The contribution with respect to distribution should be obvious from the discussions in Sections 6 and 7.

Cooperation is directly supported by alliances. Section 4 showed that alliances allow to specify arbitrary cooperation protocols and are not restricted to client-server relationships. This way alliances allow to integrate cooperative activities into WFMS besides mere procedure-oriented ones.

Flexibility Alliances are event-driven. This way the initiative for communications and actions remain with the objects. The rule-based specification of alliances cover a large amount of possible messages sequences but guarantee that dependencies between activities are not violated. On the other hand the autonomy of objects is not restricted, since the initiative for communications and actions remains with the objects. This way objects can act spontaneously, and user-interface objects — and consequently users itself — can be easily integrated. This allows users to influence the execution of workflows.

Furthermore, alliances allow for modular handling of system internal errors, especially those caused by distribution and inter-node communication. This helps to free the application code of objects from

[11] By type incompatibilities we mean syntactically different but semantically equivalent messages between caller and receiver. By interface protocol we mean object-local restrictions on sequences of invocations.

complex error handling chores. This way alliances contribute to enhance distribution transparency from the objects' point of view.

10 Conclusion and Outlook

We consider this paper to make two contributions: Distribution of workflow management systems and cooperative behavior of objects in distributed systems. We also show how these two issues can be

We proposed a layered architecture for WFMS. The main focus was on the distributed implementation of a WFMS and on the support of cooperative execution of workflows by autonomous actors. We introduced a new construct called alliance which materializes inter-object cooperation protocols in a distributed object space. We showed how workflow models can be mapped to objects and alliances and how alliances can contribute to a distributed implementation of the central components of a WFMS, its controller, notification, and object manager, and to a cooperative execution of workflows. We further demonstrated how objects and alliances can be implemented on top of a distributed object management system which on its part uses standard services and protocols.

We have developed prototypical implementation of our approach. It includes two of the four versions of the evaluation cycle outlined in Section 7: a centralized one with a fixed master and non-replicated state, and the variant with distributed control but still a centralized state. For the latter we used a majority consensus voting algorithm for active replicates as specified in [26]. The voting algorithm is used for role selection.

The prototype is based on a *CORBA* implementation. We use the persistent object management system *OBST* [49] to implement the required database services. The current prototype does not use any distributed database services, i.e., the object index is maintained centrally. Also the persistent state of alliances is not replicated across several nodes but stored in a central database. The prototype does not support fine-grained active objects by threads since the *CORBA* implementation does not support multi-threading. In order to gain experiences with fine-grained active objects we did some promising experiments with *Concurrent C/C++* [19] to implement objects and alliances apart from the *CORBA* based implementation . They proved the benefits of high-level language constructs to implement fine-grained active objects. As application example we selected the transportation company world which accompanied us throughout this paper.

If one compares alliance-based communication with "classical" non-mediated object invocation — especially if we assume a distributed implementation variant of the evaluation cycle — it is not unexpected that we have to pay for the additional functionality of alliances by performance losses. Consequently, optimization is vital. The main performance parameter is communication costs. If the distributed implementation variant is used communication costs which arise in connection with evaluating one message request depend on the number of participants, i.e. representatives of the alliance. In the centralized variant communication costs depend on the location of objects and the master, since we assume that communication via an ORB or by RPC is far more expensive than communication costs inside one process. Consequently, optimization should start with controlling the location of objects that cooperate with each other. Putting participants of an alliance and representatives together at one node can considerably reduce communication costs. Of course, this is not always possible due to the size of the objects or because of special application requirements (e.g., security). We are currently investigating what kind of distribution control primitives such as, e.g., object migration and attachment [1], are appropriate in order to extend the functionality of alliances by application-specific distribution control strategies.

A second important issue is the design of alliances. This seems a non-trivial task even for small examples. Given a declarative execution control description, a lot of additional execution semantics, especially rules for error handling, must be added to bring alliances to work. Fortunately, one may be able to rely on the experience of protocol design from the telecommunications field (e.g., [46]). We are currently looking into how transition system logics as *linear temporal logic* or *computational tree logic (CTL)*[12] can be applied to define correctness of alliance types[13], how such a specification can be systematically transformed to protocol rules, and which techniques of static analysis of alliance types can be applied to ensure their correctness.

[12] For instance, the temporal ordering condition of workflows $w_1 < w_2$ introduced in Section 2.1 can be expressed as a CTL formula.

[13] Ngu et al. recently proposed to use propositional temporal logic to specify and validate so-called interoperable transactions which can be compared with "alliances" on a conceptual (workflow) level [38].

References

[1] B. Achauer. The DOWL distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, Sep. 1993.

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

[3] 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.

[4] 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.

[5] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 99–108, 1993.

[6] André Arnold. *Finite Transitions Systems*. Prentice Hall, 1994.

[7] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifiying and enforcing intertask dependencies. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 134–145, Dublin, Ireland, Aug 1993.

[8] C. Beeri and T. Milo. A model for active object oriented database. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 337–349, Barcelona, Spain, 1991.

[9] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an open system: The REACH rule system. 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 111–126. Springer Verlag, 1994.

[10] R. G. Cattell. *Object Database Standard: ODMG – 93*. Morgan Kaufmann, 1994.

[11] 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)*, pages 606–617, Santiago de Chile, Chile, Sep 1994.

[12] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 204–214, Atlantic City, NJ, May 1990.

[13] 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)*, pages 317–326, Barcelona, Spain, 1991.

[14] J. Eder and W. Liebhart. The workflow activity model WAMO. In *Proc. of Int. Conference on Cooperative Information Systems*, pages 87–98, 1995.

[15] G. Florijn. Object protocols as functional parsers. In W. Olthoff, editor, *ECOOP'95 — Object-Oriented Programming*, volume 952 of *LNCS*, pages 351–373. Springer, 1995.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 249–259, May 1987.

[18] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. 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 23–39. Springer Verlag, 1994.

[19] N. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, 1989.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] S. Jablonski. Workflow-Management-Systeme: Motivation, Modellierung, Architektur. *Informatik Spektrum*, 18(1):13–24, Feb. 1995, (in German).

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

[27] 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.

[28] 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.

[29] A. M. Kotz, K. R. Dittrich, and J. A. Mülle. Supporting semantic rules by a generalized event-trigger mechanism. In *Proc. Intl. Conf. Extending Database Technology (EDBT)*, pages 76–91, Venice, Italy, Mar. 1988.

[30] D. Lea and J. Marlowe. Interface-based protocol specification of open systems using PSL. In W. Olthoff, editor, *ECOOP'95 — Object-Oriented Programming*, volume 952 of *LNCS*, pages 374–398. Springer, 1995.

[31] F. Leymann. Supporting business transactions via partial backward Recovery in Workflow Management Systems. In *Datenbanken in Büro, Technik und Wissenschaft (BTW)*, pages 51–70. Springer Verlag, 1995.

[32] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A family of order-preserving scalable distributed data structures. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 342–353, Santiago, Chile, 1994.

[33] 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.

[34] 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, 1994.

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

[36] J. C. McCarthy and W. M. Bluestein. The computing strategy report: Workflow's progress. Technical report, Forrester Research Inc., Cambridge, 1991.

[37] 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.

[38] A. HH. Ngu, R. Meersman, and H. Weigand. Specification and verification of communication constraints for interoperable transations. In *Proc. of 2nd Int. Conference on Cooperative Information Systems*, pages 15–22, Toronto, Canada, May 1994.

[39] 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.

[40] 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.

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

[42] OSF. *An Introdution to OSF-DCE*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[43] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[44] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[45] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems*. Addison-Wesley, 1995.

[46] J. M. Schneider. *Protocol Engineering: A Rule Based Approach*. Vieweg Advanced Studies in Computer Science. Verlag Vieweg, 1992.

[47] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *6th Intl. Conf. on Distributed Computing Systems*, pages 198–205, Boston, Massachussetts, 1986.

[48] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[49] J. Uhl et al. *The Object Management System of STONE – OBST Release 3.4.3*. Forschungszentrum Informatik (FZI), Karlsruhe, 1994.

[50] 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.

[51] 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.

[52] 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.

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

Contents