

# Supporting Development of Cooperative Object Information Systems with CoLaSD

Juan Carlos Cruz

IAM-University of Berne, Daedalos Consulting AG Zürich  
Switzerland  
cruz@iam.unibe.ch

**Abstract.** Cooperative Object Information Systems are systems build from objects that work together as a single system. Objects that cooperate in the realization of common tasks. Cooperative Object Information Systems are systems constrained to continuously adapt to new requirements: new objects are introduced into the systems, cooperation protocols change, etc. Building Cooperative Object Information Systems is difficult because most of the concurrent and distributed object oriented programming languages and frameworks used to build them, provide only limited support for their specification and abstraction, making them difficult to understand, modify and customize. We show in this paper how combining distributed active objects, and object oriented coordination models and languages, particularly the CoLaSD coordination model and language, we can simplify the development of Cooperative Object Information Systems, and facilitate at the same time the evolution of their requirements.

## 1 Introduction

Today organizations are moving to exploit computing systems more effectively, and this is giving rise to a new class of large-scale distributed systems. These “Enterprise Information Systems” offer highly integrated, highly reliable computing to users who may be physically separated by large distances. These information systems combine large number of independently executing programs that cooperate as a whole, and often provide services that are critical to the organization. These cooperative information systems are strongly constrained to continuously adapt to new computational requirements: new software pieces are introduced into the systems, and the way in which those pieces cooperate may change. While different kinds of technologies have emerged to promote development and implementation of these kinds of information systems, one technology appears as the most promising today: Object Oriented Technology (OOT in the following). OOT promotes reusability, adaptability, and evolution of information systems. Although the importance of Cooperative Object Information Systems today, existing concurrent and distributed object oriented programming languages and frameworks used to build them, provide only limited support for their specification and abstraction [1], making those systems difficult to understand, modify and customize. To our point of view a possible solution to this problem goes through

the introduction of the so-called coordination models and languages into the development of Cooperative Object Information Systems. Coordination technology addresses the construction of open, and flexible systems from active and independent software entities in concurrent and distributed systems. We propose in this paper to introduce object oriented coordination models and languages into the construction of Cooperative Object Information Systems, particularly we propose to use CoLaSD, a coordination model and language [3][4][5] for distributed active objects. The CoLaSD coordination model and language is based on the notion of Coordination Groups, entities that specify and enforce coordination within groups of collaborating distributed active objects. Coordination groups are specified independently of the internal representation of the distributed active objects they coordinate. This separation of concerns promotes design of Cooperative Object Information Systems with higher potential for reuse, it facilitates their abstraction, their understanding and their evolution. Furthermore, CoLaSD support dynamic evolution of the coordination: the coordination behavior can be modified, Coordination Groups can be created and destroyed dynamically, and distributed active objects can join and leave the Coordination Groups.

This paper is organized in the following way: Section 2 introduces the CoLaSD coordination model and language. Section 3, goes into the details of the model by showing how to build a Distributed Agenda system [10]. And, finally, Section 4, compares the CoLaSD model with respect to related work.

## **2 The CoLaSD Coordination Model and Language**

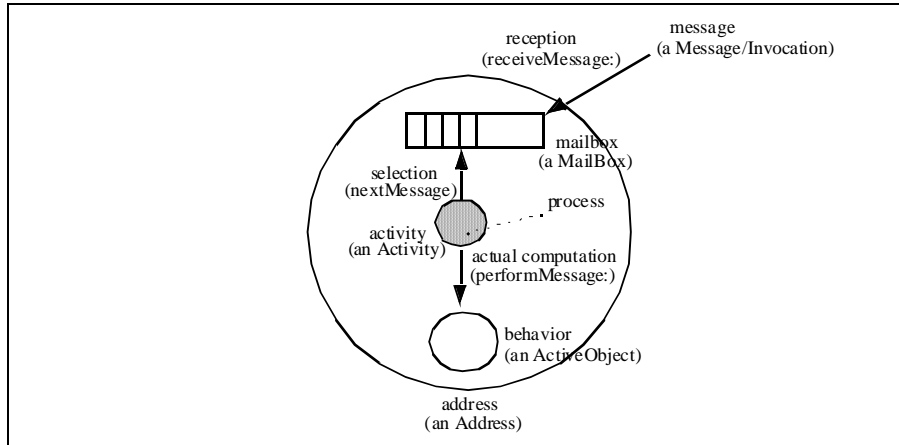
The CoLaSD coordination model and language is built out two kinds of entities: the Coordination Groups and the Group Participants (participants in the following).

### **2.1 The Participants**

Participants in CoLaSD are atomic distributed concurrent active objects (Fig.1) (distributed active objects in the following). They are concurrent because they process multiple method invocations at the same time. They are active because they have control over the concurrent methods invocations. They are distributed because physically they run on different processors or machines. And, they are atomic because they process invocations atomically. In CoLaSD participants are subclasses of the class COLASDDistributedObject. This special class manages transparently all aspects related with the internal activity of distributed active objects and their interaction with the CGs.

Distributed active objects in CoLaSD communicate by exchanging messages asynchronously. Messages represent requests for methods invocations. Replies to method invocations are managed using implicit Futures. When an object expects a reply from a method invocation on another object, it is not forced to wait until it tries to retrieve it by using the value of the future (**result** construct). The object may

ask if the result has arrived (`isReady` construct) or decide to wait for it anyhow (`wait` construct).



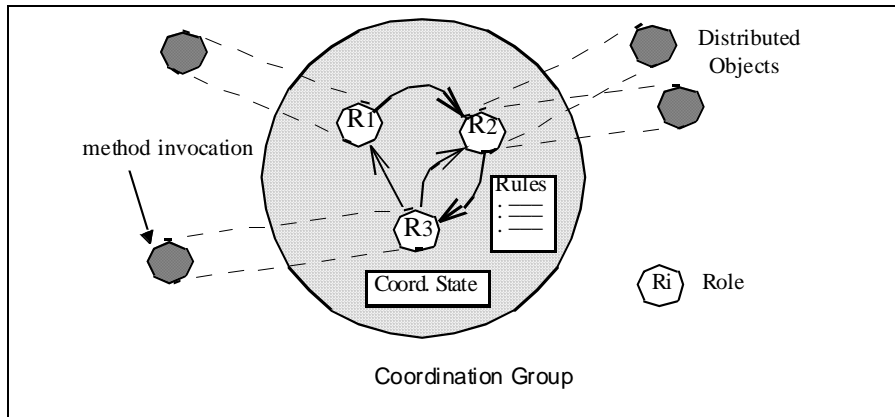
**Fig. 1.** Components of distributed active objects

There are three types of asynchronous messages in CoLaSD: `apply`, `call` and `send` messages (ACS). The ACS messages define three different ways to manage failures occurred during remote methods invocations. These three types of asynchronous messages come from a communication protocol introduced by [9] called ACS, a protocol designed to support reliable object distributed applications. In the ACS protocol method invocations are enclosed inside atomic actions and the logical nesting of methods invocations corresponds to the nesting of actions. All the three types of asynchronous messages `apply`; `call` and `send` imply the execution of some subaction on a remote object. An `apply` message implies the abort of the sender actions if a failure occurs during the execution of the subaction on the remote object. A `call` message does not imply necessarily the abort of the sender actions if a failure occurs during the execution of the subaction on the remote object. Different subactions on remote objects are thus allowed to fail independently of each other. The sender may know that the request has failed, and may consequently choose to abort or to continue the execution of its actions. In both `apply` and `call` messages, the sender may decide to abort its actions independently of its subactions. All the subactions are aborted if this happens. To safely break atomicity when an independent subaction has to be triggered, the ACS protocol introduces a third type of message called `send`. Senders of `send` messages, does not rely, neither expect, any reply from the execution of the subaction on the remote object.

## 2.2 Coordination Groups

A Coordination Group (CG in the following) is an entity that specifies, control and enforces the coordination of a group of distributed active objects. The primary tasks of a coordination group are: (1) to enforce cooperation actions between

participants, (2) to synchronize the occurrence of participants' actions, and (3) to enforce proactive actions (in the following proactions) on participants based on the state of the coordination.



**Fig. 2.** Coordination Groups

### Coordination Specification

A CG (Fig. 2) is composed of six elements: the Role Specification, the Coordination State, the Cooperation Protocol, the Coordination Interceptors, the Actions Synchronizations and the Proactions. We call the last four elements of a CG the behavioral specification of the CG.

- *The Role Specification:* defines the roles that participants may play in the group. A role identifies abstractly a set of entities sharing the same coordination behavior. Each role has associated a role interface.
- *The Coordination State:* defines general information needed for the coordination of the group. It concerns information like whether a given action has occurred or not into the system (i.e. historical information), or information about the state of a participant (i.e. busy or free).
- *The Cooperation Protocol:* defines cooperation actions between participants. They are associated to the roles, and specify actions that participants playing those roles must do when they receive some specific invocation requests. Participants are able to react to those invocation requests only during the time they play those roles.
- *The Coordination Interceptors:* define actions that modify the coordination state of the CG at different moments (i.e. at arrival, before execution, etc.) during the handling of an invocation request.
- *The Actions Synchronizations:* specify synchronizations constraints over invocation requests received by participants.

- *The Proactions*: specify actions that must be executed by the CG depending on the coordination state, and independently of the invocation requests received by participants.

The behavioral specification of the CG is specified using rules [2][8]. The advantage of using rules is that they make the coordination explicit. They avoid programmers to deal with the low-level details of how to realize the coordination.

### 3 An Example-A Distributed Agenda [10]

#### Problem description

The system to model is a simple collaborative agenda for members of a ‘software engineering’. The agenda assists in the management of meetings in the laboratory conference room. Several users may view and modify its contents simultaneously from their individual workstations, while preventing conflicts (i.e. planning of overlapping events). The distributed agenda allows the following services: consultation, addition, and cancellation of events. An event is structured as composed by a day, a beginning and ending time, and a comment line. To ensure a maximum reliability, the system should present a distributed architecture, there is need to guarantee consistency between the replicas of the diary. To this end, when a user modifies its copy of the diary, the other participants must accept the corresponding event modification.

#### 3.1 Role Specification

The distributed agenda CG in Fig. 3 specifies two roles: *agendas* and *members* (line 1). To play a role, a distributed active object should have at least the functionalities required by the role interface (interface compatibility). Roles can be played by more than one participant, and participants can play more than one role. Lines 2 and 3, specify the role interfaces associated to the roles *agendas* and *members*. For the role *agendas* for example, is required that the distributed active object be able to react to the invocation requests: `memberId`, `consult`, `addEvent`, and `deleteEvent`. These invocations are used in the description of the behavioral specification of the CG (lines 8, 13 and 18).

#### 3.2 Coordination State

The coordination state in a CG is specified by defining variables. They are accessible to all the participants of a CG. In the distributed agenda example we have defined two variables: `numYes` and `numNo` (line 4). These variables are used during the validation of the agenda operations. A member that modifies his or her copy of the agenda should submit his or her modifications to the other members of the CG. Validations are done by realizing a voting process, rules [4][5][6]. It is possible to associate default values (line 4) and initialization methods to the state variables.

```

1 distributedAgenda defineRoles: #(agendas, members).
2 members defineInterface: #(memberId, vote:event:).
3 agendas defineInterface: #(memberId, consult:, addEvent:, deleteEvent:).
4 distributedAgenda defineVariables: #(numYes=0, numNot=0).
5
6 [1] members defineBehavior: 'consultAgenda:aDay' as:
7 [agenda := agendas select:[:each | each memberId = receiver memberId].
8 ^agenda apply consult:aDay ].
9
10 [2] members defineBehavior: 'addEventToAgenda:anEvent' as:
11 [agenda := agendas select:[:each | each memberId = receiver memberId].
12 (receiver initiateValidation: #addEvent event:anEvent) result
13     ifTrue:[ agendas apply addEvent:anEvent ]]
14
15 [3] members defineBehavior: 'deleteEventFromAgenda: anEvent' as:
16 [agenda := agendas select:[:each | each memberId = receiver memberId].
17 receiver initiateValidation:#deleteEvent event:anEvent) result
18     ifTrue:[ agendas apply deleteEvent:anEvent ]]
19
20 [4]members defineBehavior: 'initiateValidation:operation event:anEvent' as
21 [members apply validate:anOperation event:anEvent initiator:aMember.
22 Delay forSeconds:MaxVoteDelay.
23 ^(group valueVariable:numYes) > (group valueVariable:numNot)]
24
25 [5]members defineBehavior: 'validate:anOperation event:anEvent
26     initiator:aMember' as
27 [initiator resultOf:(receiver vote:anOperation event:anEvent) ]
28
29 [6] members defineBehavior: 'resultOf:aVote' as:
30 [(aVote = 'Yes')
31     ifTrue: [ group incrVariable:numYes ]
32     ifFalse: [ group incrVariable:numNot ]]

```

**Fig. 3. Distributed Agenda**

### 3.3 Behavioral Specification

#### The Cooperation Protocol

The cooperation protocol of CG is specified by rules of the form <Role> defineBehavior: <Message> as: <Coordination Actions>, where the <Coordination Actions> represent actions that must be executed when a participant playing the role <Role> receive an invocation request <Message>. The <Coordination Actions> include actions that manipulate the coordination state, and invocation requests on other participants of the CG.

#### The Coordination Interceptors

The Coordination Interceptors have the form <Role> <Message> <Interception Point> do:<State Actions>. There are three different kinds of interceptors (InterceptAtArrival, InterceptBeforeExecution, and InterceptAfterExecution) according to the moment at which the invocation request <Message> should be intercepted. InterceptAtArrival specifies that the <State Actions> actions should be executed when the invocation request arrives to the participant. InterceptBeforeExecution and InterceptAfterExecution specify that those actions

should be executed before (and after respectively) of the execution of the invocation request by the participant. <State actions> are actions that affect exclusively the coordination state (i.e. to change the value of a state variable).

### **The Actions Synchronizations**

The Actions synchronizations have the form <Role> <Message> <Operator> <Synchronization Conditions>. They specify conditions that constraint the execution of invocation requests received by participants playing the role <Role>. Two types of operators may be specified: Ignore and Disable. Depending on the operator, the invocation requests <Message> are ignored or delayed by the participants, of course if the <Synchronization Conditions>conditions hold. To ignore an invocation request means that the invocation is simply not executed by the participant, and to delay it, that the invocation is put it into the participant's mailbox (invocations mailbox) to be processed lately. Actions Synchronizations are necessary to ensure properties such as: (1) mutual exclusion, and (2) temporal ordering of invocations. The <Synchronization Conditions> refers to information like: the identity of the receiver or the sender of the invocation request, the values of the arguments of the invocation request, and the coordination state of the CG.

### **The Proactions**

The coordination of the CG has been defined purely reactive until now. Coordination Actions are done in response to invocation requests received by CG participants. Actions cannot be initiated by the CG independently of the messages exchanged by the participants. Proactive behavior [2] is introduced in CoLaSD to specify actions that must be enforced by the CG independently of messages exchanged by participants (assuming that certain conditions holds). Proactions have the form <Proaction Conditions> do: <Coordination Actions>. The <Proaction Conditions> are conditions referring the coordination state of the CG. The evaluation of the proactions is done non-deterministically by the CG.

In the distributed agenda example (Fig. 3) six rules are defined:

*Rule 1* (line 6), *Rule 2* (line 10), *Rule 3* (line 15): defines the operations on the "distributed agenda": `consultAgenda:`, `addEventToAgenda:`, and `deleteEventFromAgenda:`. Each one of the operations triggers the corresponding invocation request on the members' agenda. Both `addEventToAgenda:` and `deleteEventFromAgenda:` operations require a previous validation of the operation (line 12 and 17).

*Rule 4*(line 20): a member that modifies his or her copy of the agenda should previously validate the modification with the other members. An invocation request `validate:event:initiator:` is sent to all the members of the CG. Each member decides to accept or not to validate the modification, and sends back a

message `resultOf:` with his or her decision. The initiator of the validation process blocks during `MaxVoteDelay` seconds to evaluate the result of the validation.

*Rule 5*(line 25): when a member is requested to validate an agenda operation. It sends a message `resultOf:` with his or her vote to the initiator of the validation process.

*Rule 6*(line 29): when the initiator of a validation process receives the results of the validation from the other members, it counts them. Positive and negative replies are counted using the variables `numYes`, `numNot`.

### 3.4 Pseudo-Variables

There are three pseudo-variables that can be used within the CGs. They are: `group`, `receiver`, and `sender`. The `group` variable refers to the CG on which the variable appears (lines 23, 31, and 32). The `sender` pseudo-variable refers to the participant that sent the invocation request, and the `receiver` pseudo-variable the participant handling the invocation request (lines 12 and 17).

### 3.5 Group Creation- The CORODS coordination service

```
1 namingService:=ORBObject resolveInitialReferences:#NameService.  
2 corodsService:=namingService contextResolve:'CORODS' asDSTName.  
3  
4 distributedAgenda:=corodsService createCGNamed: 'DistributedAgenda'.  
5 distributedAgenda defineRoles: #(...)  
6 ... /*distributed agenda specification..  
7  
8 laboratoryAgenda := corodsService      "distribute agenda instance"  
9                                createCGInstance: 'LabAgenda'  
10                               forCGNamed: 'DistributedAgenda'
```

**Fig. 4. Group Creation**

CGs are created across the network by using a coordination service called CORODS. To create a CG, a client sends the message `createCGNamed: <Group Name>` to the CORODS service (Fig. 4, line 4). The names of the CGs are unique within the service. Instances of CGs are created by sending the message `createCGInstance: <Instance Name>` to the CORODS service (Fig 4, line 8) or by sending the message `createCGInstance: <Instance Name> forCGNamed: <Group Name>`. References to CGs instances are obtained by sending the message `getReferenceToCGInstanceNamed: <Instance Name>`.

### 3.6 Dynamic Properties

```
1 distributedAgenda defineVariable: #(groupIsValidating=false)
2 [7] r7 :=members 'initiateValidation:operation event:anEvent' disable
3 [ (group valueVariable: groupIsValidating) not ]
4
5 [8] r8:= members 'initiateValidation:operation event:anEvent'
6 InterceptBeforeExecution do:
7 [ group setVariable: groupIsValidating: value: true ]
8
9 [9] r9:=members 'initiateValidation:operation event:anEvent'
10 InterceptAfterExecution do:
11 [ group setVariable: groupIsValidating: value: false]
12
13 distributedAgenda addRule: r1; addRule: r2; addRule: r3.
```

**Fig. 5.** Dynamic Properties

CoLaSD supports three types of dynamic coordination changes: (1) new participants can join or leave CGs, (2) new CGs can be created and destroyed, and (3) the coordination behavior can change by adding or removing rules to the CG. To introduce new participants a message `addParticipant: <newParticipant> forRole: <Role>` must be sent to the CG (`removeParticipant: <Participant> fromRole: <Role>` to leave it). To add a new coordination rule, a message `addRule: <aRule>` must be sent to the CG (`removeRule: <aRule>` to remove a rule). In Fig 5, we can see how rules 7, 8 and 9 are added to the CG (line 13). These new rules control that only one validation process occurs in the system at the same time. Rule [7] (line 2) specifies that whenever a validation process is running additional validation messages should be disable. A variable called `groupIsValidating` is used to do this.

## 4 Related Work and Conclusions

Concurrent and distributed object oriented languages provide only limited support for the specification an abstraction of Cooperative Object Information Systems. In those systems the objects that compose the systems and the way in which they are composed founds mixed into the objects' code, making them difficult to understand, modify and customize. The idea of separating coordinational and computational aspects of systems using coordination languages [7] is an interesting approach to simplify the development of these kinds of systems. CoLaSD is a coordination model and language that supports coordination of Cooperative Object Systems. It promotes design of Cooperative Object Information Systems with higher potential for reuse, it facilitates their abstraction, their understanding and their evolution. Due to space limitation we limit this related work to coordination languages specifically designed for object oriented systems and to coordination languages which use the same approach of message interception to realize the coordination (for a complete related work refer to [3][4][5]). The most important related works are Synchronizers [6] and Moses [8]. Both specify coordination using rules as in CoLaSD. The main differences with respect to CoLaSD are: (1) In synchronizers the coordination is restricted to synchronization of invocation request, in CoLaSD

and Moses coordination actions can be enforced on participants too. Nevertheless, in Moses those actions only affect the receiver of the invocation requests. In CoLaSD coordination actions may affect any participant of a CG. (2) Synchronizers are pure reactive entities, they react to the arrival of invocation requests. In both Moses and CoLaSD proactions may be initiated independently of the arrival of invocation requests. (3) Synchronizers, neither Moses support dynamic evolution of the coordination as in CoLaSD. In CoLaSD the coordination can be modified dynamically: coordination rules are added or deleted, distributed objects join or leave CGs, and CGs are created and destroyed. Finally, (4) Synchronizers neither Moses includes the possibility of failures into their models.

### References

1. M.Aksit, and L. Bergmans, *Obstacles in Object-Oriented Software Development*, OOPSLA'92, vol. 27, no. 10, October 1992, pp. 341-358.
2. J-M.Andreoli, S. Freeman and R.Pareschi, *The Coordination Language Facility: Coordination of Distributed Objects*, TAPOS , vol.2, no. 2, 1996, pp. 635-667.
3. J.C.Cruz and S. Ducasse, *A Group Based Approach for Coordinating Active Objects*, Coordination'99, LNCS 1594, Springer Verlag, pp. 355-370.
4. J.C.Cruz and S. Ducasse, *Coordinating Open Distributed Systems*, Future Trends of Distributed Computing Systems '99, IEEE, pp. 125-130.
5. J.C.Cruz, CORODS: a Coordination Programming System for Open Distributed Systems, Languages and Models of Objects' 2001, L'object vol. 7, no. 1-2/2001.
6. S. Frolund, *Coordinating Distributed Objects*, MIT Press, 1996.
7. D. Gelernter and N. Carriero, *Coordination Languages and their Significance*, Communication of the ACM vol. 35, no. 2, February 1992.
8. N.Minsky and V. Ungureanu, *Regulated Coordination in Open Distributed Systems*, COORDINATION'97, LNCS 1282, Springer-Verlag, 1997, pp. 81-97.
9. R.Guerraoui, R. Capobianchi, A. Lanusse and P. Roux, Nesting Actions thorough Asynchronous Message Passing: the ACS protocol, ECOOP 92.
10. R. Bastide and Didier Buchs, *Model, Formalisms and Methods for Object-Oriented Distributed Computing*, ECOOP-Reader, 1998.