A Group Based Approach for Coordinating Active Objects

Juan Carlos Cruz, Stéphane Ducasse[1]

Abstract. Although coordination of concurrent objects is a fundamental aspect of object-oriented concurrent programming, there is only little support for its specification and abstraction at the language level. This is a problem because coordination is often buried in the code of the coordinated objects, leading to a lack of abstraction and reuse. Here we present CoLaS, a coordination model and its implementation based on the notion of Coordination Groups. By clearly identifying and separating the coordination from the coordinated objects CoLaS provides a better abstraction and reuse of the coordination and the coordinated objects. Moreover CoLaS's high dynamicity provides better support for coordination of active objects.

1 Introduction

Coordination technology addresses the construction of open, flexible systems from active and independent software entities in concurrent and distributed systems. Although coordination is a fundamental aspect of object-oriented programming languages for concurrent and distributed systems, existing object-oriented languages provide only limited support for its specification and abstraction [Frol93a, Aksi92a]. Furthermore, in these languages it is not possible to abstract coordination patterns from the representation of the coordinated objects. Coordination policies are generally hard-wired into applications making them difficult to understand, modify and customize. This is a serious problem when developing open and flexible systems. In those systems the coordination policies need to be adapted dynamically to respond to new coordination requirements.

In this paper we introduce a coordination model called CoLaS based on the notion of *coordination groups*. The CoLaS coordination model is based on the specification and *enforcement* of cooperation protocols, multi-action synchronizations, and proactive behaviour within groups of collaborating active objects. The current version of CoLaS is implemented in Smalltalk on top of the Actalk platform [Brio96a].

Coordination groups are high-level abstractions for managing the coordination aspect in concurrent object-oriented systems. We roughly define a *coordination group* as a set of policies that regulates the activities of a group of active objects — called *participants*. Groups are specified independently of the internal representation of their participants. This independence allows for a clear separation of computation and coordination concerns (as promoted by coordination languages [Gele92a]). Separation of concerns promotes design with a greater potential for reuse. Active objects may be reused independently of how they are coordinated, and coordination patterns can be reused in-

1. *Author's address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.3315. *Fax:* +41 (31) 631.3965. *E-mail:* {cruz, ducasse}@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~cruz.

dependently on different groups of active objects. Coordination groups support dynamic evolution of coordination in three distinct axes: (1) groups are created dynamically at any time; (2) participants join and leave the coordination groups whenever they want, and (3) their behaviour can be modified dynamically to adapt to new coordination requirements.

This paper is organized as follows: Section 2 discusses existing problems in the realization of the coordination in software systems, and establishes a list of requirements for an ideal coordination language for active objects. Section 3 introduces the CoLaS coordination model. Section 4 goes into the details by describing how CoLaS proposes a solution to the classical Gas Station example [Helm85a]. Section 5 presents some dynamic aspects of the approach. Section 6 illustrates proactive behaviour using the Electronic Vote example [Mins97a]. Section 7 evaluates the CoLaS model with respect to the problems and requirements defined in Section 2. Finally Section 8 concludes with a discussion evaluating our contributions compared with related work.

2 Language Support for Coordination in COO Systems

We consider that the primary tasks of coordination in concurrent object systems (COOs) are: (1) to support the creation of active objects, (2) to enforce cooperation actions between active objects, (3) to synchronise the occurrence of those actions in the system, and (4) to enforce proactive behaviour [And96b] on the system based on the state of the coordination. Providing a *high level* construct for explicitly specifying coordination separately from the computation and supporting dynamic evolution of the requirements addresses the following common problems:

No separation of computational and coordination concerns. In most concurrent object systems coordination policies are hard-coded into the actions of the cooperating objects [Frol93a, Lope97a] making understanding, modification and customisation difficult. This lack of separation of concerns promotes design with poor potential for reuse [Aksi92a]. In those systems objects cannot be reused independently of how they are coordinated and coordination patterns cannot be reused independently on different groups of objects.

Lack of high level coordination abstractions. Existing concurrent object-oriented languages (COOLs) offer low level support for the expression and abstraction of complex object cooperations and large scale synchronizations involving more than just a pair of objects [Aksi92a]. For example, in Java coordination can be modelled at a very low level of abstraction: threads model asynchronous activities; the synchronized keyword, the wait, notify and notifyAll methods are used to coordinate activities across threads. While the set of provided constructs can be used to solve non trivial coordination problems, in practise only expert programmers are able to handle them appropriately. Java programmers tend to rely on design patterns [Lea96a] to solve common coordination problems.

Dynamic Evolution of Coordination. The fact that coordination is mixed within the application code makes the coordination evolution difficult to realize. Indeed, three main changes in a coordination group can impact it: (1) the *creation* of new coordination groups, (2) the *addition/removal of new participants* to a coordination group (i.e. new objects come into play or leave the application), and (3) the *addition/removal of coordination policies*. The changes range broadly from local redefinition and recompilation of coordination and/or participants to the overall system redefinition and recompilation

The integration of coordination into a COOL should propose a solution to these problems. In the following we elaborate on the requirements for such an ideal language.

2.1 Requirements for a Coordination Language for Active Objects

Coordination Specification. Are the coordination policies fixed within the system? Can coordination policies be incrementally specified? Is the coordination expressed declaratively or procedurally?

It must be possible for programmers to define new coordination policies [Mins97a] within the system, their specification should be *user-defined*. Contrary to Synchronizers [Frol93a] that do not support incremental definition of the synchronization policies, the coordination policies should be defined *incrementally* from others like in [Aksi94a, Mukh95a, Duca98c]. Finally, as proposed in [Frol93a, Andr96b, Mins97a] policies should be *declarative* to avoid programmers deal with low-level details on how the coordination occurs.

Coordination Properties. Is the coordination: data-driven or control driven [Arbad96b]? Transparently integrated in the host languages? Non-intrusive? Is the coordination centralized (i.e. objects are coordinated using a central coordinator agent), decentralized (i.e. objects communicate explicitly to realize the coordination) or hybrid (i.e. achieved through the cooperation of both the objects and a coordinator agent) [Mukh95a]?

As COOLs promote data encapsulation and behaviour over data, the coordination in COOs must be *control driven* [Frol93a, Mukh95a, Arba96b, Mins97a].

Contrary to Linda based approaches [Kiel96a] where the coordinated objects are aware of the virtual shared space to which they communicate, coordination should be *transparent* from the point of view of the coordinated objects [Frol93a, Mukh95a, Mins97a]. Moreover, it should be *non-intrusive*: based on public interface of the coordinated object and not relying on their internal data.

Finally, the coordination must be based on a *hybrid* model [Frol93a, Aksi94b, Mukh95a, Mins97a]. The problem with centralized models [Agha93c, Andr96b] is that objects are forced to interact with a coordinator agent, and with decentralized models [Papa96a] is that objects must know other objects to realize coordination. The reusability of objects and coordination is limited in both cases.

Coordination Behaviour. Is coordination limited to synchronization of actions; or Can actions be enforced [Duca98c] and/or initiated [Andr96b] by the system? What kinds of information should be referred to by the coordination policies [Bloo79a]?

Coordination should not be limited (as in [Frol93a,Aksi94a]) to the synchronization of messages, it should be possible to enforce actions in coordinated objects as a reaction to certain messages received by the objects. Moreover, it should be possible to initiate actions in the system (i.e. proactive actions) depending on the state of the coordination [Andr96b, Mins97a]. The coordination state should take into account the state of the coordinated objects [Papa96a] and the history of the coordination.

Evolution. Can coordination policies be created and/or modified dynamically? Do coordination policies support the addition and removal of coordinated objects? Can we define new coordination patterns dynamically?

The coordination should be highly *dynamic*: objects must be able to join and/or leave the coordination at any time, coordination policies must be modifiable on the fly, and new coordination patterns must be able to be created at run-time [Andr96b]. A highly dynamic system will be able to respond to new coordination requirements.

Formal Properties. Can we prove that the behaviour of an object is compatible with the coordination policies of the system? Can we prove that the coordination will develop correctly (i.e. safe)?

Ideally we would like to have a formal model fully integrated to the coordination language that checks the ability of objects to be coordinated. Furthermore, we would like to be able to prove certain safety and liveness properties of the coordination like deadlock freeness, termination, etc. The formal model should not be limited to the specification and the verification of the coordination as in [Alle94c] but causally connected to the language in the sense of “executable specification”.

A Quick Overview of CoLaS. According to these requirements, CoLaS is a *hybrid* model that supports *user-defined explicit* and *non-intrusive* object group coordination based on the *transparent* synchronisation and *enforcement* of exchanged messages. Moreover, coordination is not limited to coordinator state. CoLaS supports the *dynamic* evolution of the coordination.

3 The CoLaS Coordination Model

We propose a coordination model for COOs based on the notion of coordination groups. A coordination group specifies, encapsulates and enforces the coordination of a group of cooperating concurrent objects. The CoLaS model is built out of two kind of entities: the participants and the coordination groups.

3.1 Participants

In CoLaS the participants are *active objects* [Briot96a]: objects that have control over concurrent message invocations. Incoming messages are stored into a mailbox until the object is ready to process them. By default, an active object treats its incoming messages in a sequential way. In CoLaS active objects communicate by exchanging messages concurrently in an *asynchronous* way. Replies are managed using *explicit futures* so the objects are not blocked while waiting for their replies.

3.2 Coordination Groups

A *Coordination Group* (group in the following) is an entity that specifies and enforces the coordination of a group of participants to perform a common task. According to our notion of coordination the primary tasks of a group are: (1) to enforce cooperation actions between participants, (2) to synchronize the occurrence of those actions, and (3) to enforce proactive actions [Andr96b] (in the following proactions) in participants based on the state of the coordination.

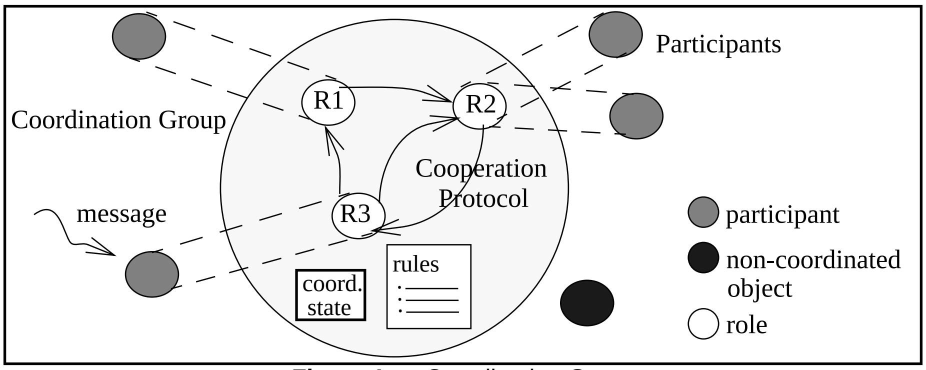

Figure 1 a Coordination Group

Coordination Specification. A group is composed of five elements: a Role Specification, a Coordination State, a Cooperation Protocol, Multi-Action Synchronizations and Proactions (Fig. 1).

- *The Role Specification:* defines the roles that participants may play in the group. Similar to connector roles [Alle94c], a role identifies abstractly entities sharing the same coordination behavioural specification within the group. Each role has an associated role interface.
- *The Coordination State:* defines information needed for the group coordination. It is global to the group and/or local to each participant in a given role.
- *The Cooperation Protocol:* defines implications between participant actions (e.g. the treatment of a message implies some other actions).
- *The Multi-Action Synchronizations:* specifies synchronisation constraints over messages exchanged by participants.
- *The Proactions:* specifies actions that must be executed by the group depending on the coordination state, independently of the messages exchanged by the participants.

The last three elements are specified using rules [Andr96b].

Object Group Participation. Objects join groups by enrolling to group *roles*. To play a role in a group, an object should possess at least the functionalities required by this role (interface compatibility). A role can be played by more than one object. Objects join and leave a group at any time without disturbing other participants.

Coordination Enforcement. When a participant handles a message waiting in its mailbox, the group checks if cooperation and/or multi-action synchronisation rules apply to this message. If so, the group enforces them (e.g. sends new messages, forbid others, etc.). Synchronisation rules are verified prior to the cooperation rules. They constrain the execution of the message and the actions specified in the cooperation rules. In contrast, as proaction rules do not depend on messages but on the state of coordination they are repeatedly checked. In every case, the group guarantees the consistency of participants during the enforcement process.

4 A Detailed View of the CoLaS Model

To help in understanding CoLaS, we present the classical Gas Station example [Helm85a]. Using this example we show the different elements of our model as well as their characteristics. This example illustrates the following coordination problems:

- *Transfer of information between entities:* car drivers communicate with cashiers to get authorizations to take gas from pumps. Pumps receive authorizations from cashiers to give fuel to drivers. Money and gas representations flow between participants.
- *Multi-action synchronizations:* there are different synchronisation constraints that have to be respected: cashiers must not authorize pumps to give fuel before being paid by drivers. Pumps must not give fuel to drivers before being authorized by cashiers.
- *Management of access to shared resources:* Cashiers must not authorize more drivers to take gas than there are pumps. They must prevent several drivers from getting gas from the same pump at the same time because there is only one hose per pump.
- *Dynamic evolution of the coordination.* New participants can leave or join the system (i.e. when a pump malfunctions or when new car drivers want to take gas).

4.1 A Case Study - The Gas Station System

Problem Description. The gas station consists of *p* pumps where car drivers can take gas for their cars. At a particular moment in time *n* car drivers can come to the gas station to obtain gas, but only *p* of them will be served at the same time. This is because each pump has only one hose to discharge gas. A car driver first has to pay an amount of money to one of the *m* cashiers. Then, the cashier orders a free pump to prepare to pump fuel. The driver receives an authorization from the cashier to take his gas, and finally he takes the amount of gas he paid for. In Fig. 2 we show the UML description of the different participants of the gas station as well as the interaction diagrams that describes the different cooperation actions that should be enforced during the coordination. Both actions pay(amount) and takeHose(pump) correspond to actions initiated by car-drivers. Note that this example does not really reflect reality: the cashier should be considered as an automatic machine but we follow the original example [Helm85a].

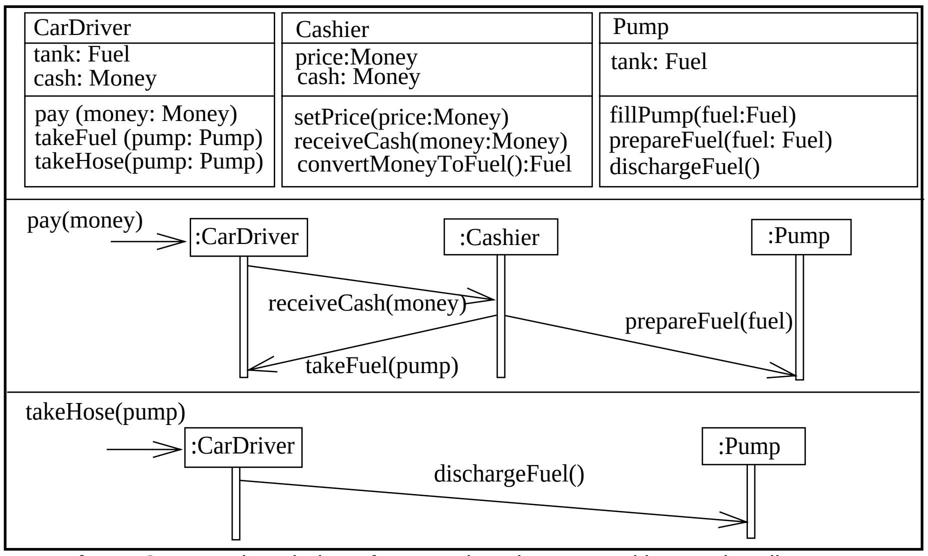

Figure 2 UML description of gas station classes, and interaction diagrams of the cooperation actions

4.2 Role Specification

In the gas station example, participants play one of the following roles: car-drivers, cashiers and pumps. Roles in a group are defined by sending the message defineRoles: to a coordination group (line 2 in Fig. 3). The minimal interface that an object should support to play a role is specified sending the message defineInterface:forRole: (e.g. an object that participates as a cashier should at least understand the messages receive-Cash: and convertMoneyToFuel:replyTo: line 4 in Fig. 3).

4.3 Coordination State

The coordination state of a group is specified by declaring variables. It is global to the group (*group coordination variables*), and/or local to each participant (*per-role coordination variables*) in a role. Coordination state variables are created using the messages:defineVariable:initialValue: and/or defineParticipantVariable:forRole:initialValue: respectively (lines 7 and 8 in Fig. 3). In the example, the per-role variable represents the state of a pump (busy/free).The variable isFree is used to guarantee exclusive access of drivers to pumps.

```
1.   gasStation := CoordinationGroup new name: 'gas-station'.
2.   gasStation defineRoles: #('drivers' 'cashiers' 'pumps').
3.   gasStation defineInterface:#('pay:' 'takeFuel:' 'takeHose:') forRole: 'drivers'.
4.   gasStation defineInterface: #('receiveCash:' 'convertMoneyToFuel:replyTo:')
5.                   forRole: 'cashiers'.
6.   gasStation defineInterface: #('prepareFuel:' 'dischargeFuel')  forRole: 'pumps'.
7.   gasStation defineVariable:'whichPump' initialValue: nil.
8.   gasStation defineParticipantVariable: 'isFree' forRole: 'pumps' initialValue: true.
```

Figure 3 Role Specification and Coordination State of the Gas Station

4.4 Coordination Behaviour Specification.

The coordination, i.e. the cooperation protocol, the multi-action synchronizations and the proactions, is specified using two different types of rules [Andr96b, Mins97a, Duca98c] as follows:

Rule1	= <Message> <Operator> <Clauses>
Message	= <Role> <MethodSelector> <Arguments>
Operator	= ImpliesBefore | ImpliesAfter | Disable | Ignore | Atomic
Rule2	= <Condition> < ProOperator> <Clauses>
ProOperator	= Once | Always

In Rule1 <Message> describes the message being treated (the symbol “*”can be used to specify any message). <Operator> specifies the semantics of the <Clauses>. Indeed for the operators ImpliesBefore or ImpliesAfter that specify a cooperation protocol, <Clauses> are possible *coordination actions* (like sending a message to another participant, changing the coordination state, etc.) that occurs before or after the message execution. For the operators Disable, Ignore and Atomic that specify multi-action synchronizations, <Clauses> are *synchronisation conditions* (referring to the state of the group, or to message information like the sender/receiver or arguments) that constrain the message execution (depending on the operator the message is ignored or delayed). For the Atomic operator, <Message> represents a set of messages that must be executed at the same time (i.e. those messages can refer to messages in different roles). In Message <Role> represents a role of the group and <MethodSelector> <Arguments> a message.

Rule2 specifies proactions [Andr96b]. For <ProOperator> operators Once and Always, the <Clauses> are the actions that are performed when the <Condition> referring to the state of the coordination holds. We explain these operators later in this paper. In CoLaS rules are added to a group by sending the message addRule:<Rule>.

4.4.1 Cooperation Protocol Specification

```
[r1] drivers pay: money  ImpliesAfter [ cashiers receiveCash: money ]

[r2] cashiers receiveCash: money  ImpliesBefore [
          | quantityReply pump |
          quantityReply := CoordFuture new.
          receiver convertMoneyToFuel: money replyTo: quantityReply.
          pump := group selectAParticipantWithRole: ’pumps’
                      that: [ :participant | valueVariable: 'isFree' ofParticipant: participant ].
          group setVariable: 'isFree' ofParticipant: pump value: false.
          group setVariable: ’whichPump’ value: pump.
          pump prepareFuel: ( quantityReply getValue).
          sender takeFuel: pump ]

[r3] drivers takeHose: pump ImpliesAfter [ pump dischargeFuel ]

[r4] pumps dischargeFuel ImpliesAfter [
          group setVariable: ’isFree’ ofParticipant: receiver value: true ]
```

Figure 4 Cooperation Specification

A cooperation protocol is specified by a set of implication rules (operators ImpliesBefore and ImpliesAfter) that define *coordination actions* that must be triggered before or after the execution of a message. In the gas station example Fig. 4, four implication rules are defined:

Rule 1 (line 1): The driver invokes his method pay: specifying the amount of money of gas he wants to buy. This triggers a receiveCash: action at the cashier.

Rule 2 (line 3): The cashier receives a receiveCash: message from a driver. This triggers a set of actions. First, the cashier converts the money into an amount of gas (line 6). Then the group selects a free pump (line 7) and generates two messages: one to the selected pump to prepare to give fuel (line 11) and another to the driver to indicate that he can take his gas (line 12). Additionally the state of the pump is changed to indicate it is busy (line 9).

Rule 3(line 14): The driver invokes its method takeHose indicating he will take gas from a pump. This triggers a dischargeFuel: action to the pump.

Rule 4(line 16): The pump receives dischargeFuel: from a driver. After the gas is discharged the group coordination state is updated. The state of the pump is set to free.

Coordination Actions. As shown in the above rules, the coordination actions are the following:

- Manipulations of the coordination state (rules 2 and 4). Their value is accessed (resp. assigned) using valueVariable: (resp. setVariable:value:) for the group coordination variables and valueVariable:ofParticipant: (resp. setVariable:value:ofParticipant:) for per-role coordination state variables.
- Evaluation of participant state-predicates [Papa96a]: state predicates are used to check conditions based on internal participant state. These kinds of actions are not used in the gas-station example.
- Selection of a participant: a participant playing a role is selected by sending the message selectAParticipantWithRole: <aRole> that: <predicate> (line 7) to the group. This expression returns non-deterministically a participant playing the role <role> satisfying the condition <predicate>.
- Sending an *asynchronous message* to a participant: By default messages are sent asynchronously to participants (lines 11 and 12). If the receiver represents a role the message is multicasted to each object playing that role.
- Sending a *synchronous recursive message* to a participant: As in Actalk [Briot96a] where an active object may send a synchronous message to itself by using the pseudo-variable self. In ColaS the pseudo variable receiver is used to send a *synchronous message* to the receiver of the message triggering the rule (line 6).

4.4.2 Multi-Action Synchronizations

While the cooperation part of the group defines the cooperation protocol, multi-action synchronizations define constraints on how these cooperation actions occur. When a participant wants to treat a message the group verifies if synchronisation rules apply to this message. Depending on the rule semantics and the value of the conditions associated with the rules, messages are ignored or delayed. Because of the non-determinism in which participant actions may occur in the concurrent system, multi-action synchronisation constraints are necessary to ensure properties such as: (1) mutual exclusion, (2) temporal ordering and (3) atomicity of invocations processed by the group.

In CoLaS these three types of synchronizations are specified by combining rules using the operators Ignore, Disable and Atomic.

- Invocation exclusions are specified using Ignore and Disable operators. They ensure that if a certain *synchronisation condition* is not satisfied a message is ignored (i.e. not processed) or delayed. In Fig. 5 line 1 the variable isFree is used in a Disable rule to control exclusion of receiveCash: invocations from drivers when all pumps are busy.
- Temporal ordering is based on the past invocation history. The combination of coordination state variables keeping coordination historical information and Disable based rules allow one to express that invocations occur at the right time.
- Atomic based rules ensure the indivisible execution of messages in multiple objects. While the synchronisation condition is not satisfied and not all the messages in the atomic rule are ready to be processed, those message are delayed.

```
1.    [r5] cashiers receiveCash: money Disable: [
2.              | aParticipant |
3.              aParticipant := group selectAParticipantWithRole: 'pumps'
4.                                          that:  [:participant | group valueVariable: 'isFree'
5.                                                                        ofParticipant: participant ].
6.              (aParticipant isNil) ]
```

Figure 5 Multi-Action Synchronisations

Synchronisation Conditions. Inspired by [Bloo79a] CoLaS synchronisation conditions refer to the following information. Note that they are basically state queries whereas coordination actions in the cooperation protocol are state queries and changes (message sending and coordination variable assignment).

- the invoked message (its arguments, its sender, its receiver) accessed using the predefined variables msg, arguments, sender, and receiver lines (6,12,17 Fig. 4).
- the coordination state of the group and the coordination state of each participant as shown in coordination actions description section (line 4 Fig. 5).
- the keyword true is used to specify rules without conditions.
- the current time in the system: Time information is used to determine the relative order of invoked messages in participants. The current value of the time is obtained by sending the message now to the group (i.e. a centralized notion of time).
- historical information: historical information concerns information about whether a given action has occurred or not. This information differs from synchronisation state information in that it refers to actions that are already completed, as opposed to those still in progress. Historical information are stored using group and/or per-role coordination variables. In Fig. 4 line 9 the per-role variable isFree is used to identify pumps already assigned to pump fuel.

4.4.3 Proactive Behaviour

Until now the coordination of the system has been purely reactive. *Coordination Actions* are done in response to the treatment of a message. But they cannot be initiated on their own. To introduce proactive behaviour [Andr96b] CoLaS supports proactions that ensure that certain Coordination actions are carried out by the group at a certain time, assuming that a certain coordination condition holds at that time. Two kinds of proactions are specified by the operators Once and Always. Once ensures that proactions are executed only one time when the condition is satisfied. Always ensures that the proactions are executed each time the condition is satisfied. The evaluation of the conditions is done periodically by the group.

4.5 Participants: Creation and Enrollment in Groups

After having specified a group we present how active objects are created and then how they join the groups.

```
1.    | drivers cashier pumps |
2.    drivers:= OrderedCollection with: (CarDriver new name: 'Dr1')
3.                                with: (CarDriver new name: 'Dr2')
4.                                with: (CarDriver new name: 'Dr3').
5.    cashier := Cashier new.
6.    pumps := OrderedCollection with: (Pump new name: 'P1') with: (Pump new name: 'P2').
7.
8.    gasStation addParticipants: drivers  withRole: 'drivers'.
9.    gasStation addParticipant: cashier withRole: 'cashiers'.
10.   gasStation addParticipants: pumps  withRole: 'pumps'.
```

Figure 6 Creation of participants and enrollment

Creation. In CoLaS active object classes are subclasses of the class ActiveObject. This special class manages transparently to programmers all aspects related with the internal activity of objects and their interaction with a group. Car drivers, cashiers and pumps are represented by classes Driver, Cashier, and Pump. Instances of these classes are created in lines 2, 5 and 6 in Fig. 6.

Enrollment. Active objects participate in a groups by playing a given role. An active object can join a group at any time by sending a message addParticipants:withRole: to a group. To enrol an object have to respect some interface obligations associated with the role. These obligations guarantee to the group the capacity of the object to play a such role in the cooperation enforced inside the group. They are verified during the enrollment process.

5 Dynamic Aspects

CoLaS supports three types of dynamic coordination changes: (1) new participants can *join or leave* the group at any time (as shown in Fig. 7), (2) new groups can be *created and destroyed* at any time, and (3) the *coordination behaviour* can be changed by adding and removing rules to the group (lines 1, 3, and 8 Fig. 8).

```
1.       (p := pumps withName: ’P1’ ) outOfOrder.
2.      gasStation removeParticipant: p.
3.
4.      gasStation addParticipant: (Driver new withName: ’Dr4’)  withRole: 'drivers'.
```
Figure 7 Dynamic addition and removal of participants

New Members Joining The Group. As shown below new participants can join or leave a group at any time. When a pump malfunctions, drivers should not use it any more, so it is removed from the group (line 2 Fig. 7). Line 4 shows how a new driver is added to the group. Both operations are done transparently to the group without modifying the coordination behaviour.

Coordination behaviour changes-Managing Races. The rules 1, 2 and the rules 3,4 (Fig. 4) form two sets of cooperation actions. The first group handles the payment of the gas, and the notification and preparation of drivers and pumps. The second group the pumping of the fuel. Drivers can decide when they want to pay, and when they want to take their gas. In this solution it is possible that a driver pays before another, but that the other takes the hose before, thus getting the amount of gas purchased by the driver. In Fig. 8 we modify dynamically the coordination state and policies to avoid this race condition. To prevent it we define a new per-role variable called whoPayed in which we store the identification of the driver authorized to serve gas from that pump rule 5 (line 6). We include a Ignore synchronization rule (line 8) to ignore dischargeFuel invocations coming from drivers not authorized to take gas from that pump.

```
1.      gasStation defineParticipantVariable: 'whoPayed' forRole: 'pumps' initialValue: nil.
2.
3.      [r6] cashiers receiveCash: money  ImpliesAfter[
4.              | pump |
5.              pump := group valueVariable:’whichPump’.
6.              group setVariable: ’whoPayed’ ofParticipant: pump value: sender ]
7.
8.      [r7] pumps dischargeFuel
9.              Ignore: [
10.             | whoPayed  |
11.             whoPayed := group valueVariable: 'whoPayed' ofParticipant:  receiver.
```
Figure 8 Managing Races

6 Illustrating Proactive Behaviour - The Electronic Vote

Now we illustrate how CoLaS supports fairness in the Electronic Vote example proposed by [Mins97a]. With this example we show how proactive behavior is used in CoLaS to solve a coordination problem.

Problem Description. Assume that there is a specific issue on which an open group of participants is asked to vote. Every participant in the group can initiate a vote on any issue he chooses. Each voter *may* actually vote by sending a result of his vote back to the initiator. The system must guarantee that the vote is fair: (1) a participant can vote at most once, and only within the time period allotted for this vote, (2) the counting is done correctly, and (3) the result of the vote is sent to all the participants after the deadline expiration.

Coordination Specification. In the electronic vote example Fig. 9, four rules define the coordination: rules 1 and 2 define cooperation rules, rule 3 a multi-action synchronization rule, and rule 4 a proaction rule.

```
1.   | adminVote |
2.
3.   adminVote := CoordinationGroup new name: 'electronic-voting'.
4.   adminVote defineRoles: #('voters').
5.   adminVote defineVariables: #('deadline' 'numYes' 'numNot') initialValues: #(0 0 0).
6.   adminVote defineParticipantVariable: 'hasVoted' forRole: 'voters' initialValue: false.
7.   adminVote defineInterface: #('startVote:deadline:' ' voteOn:initiator:'
8.                              'opinion:replyTo:' 'resultOf:' 'sendFinalResult:')
9.            forRole: 'voters'.
10.
11.  [r1] voters startVote: issue deadline: aDeadline ImpliesAfter: [
12.       group setVariable: 'deadline' value: aDeadline.
13.       voters voteOn: issue initiator: receiver ]
14.
15.  [r2] voters resultOf: vote ImpliesAfter: [
16.       (vote = 'Yes' )
17.            ifTrue: [ group incrVariable: 'numYes']
18.            ifFalse: [ group incrVariable: 'numNot'].
19.       group setVariable: 'hasVoted' ofParticipant: sender value: true. ]
20.
21.  [r3] voters resultOf: vote Ignore: [
22.       (group valueVariable: 'deadline' < TIme now )
23.            or: [ group valueVariable: 'hasVoted' ofParticipant: sender]]
24.
25.  [r4]  (Time now > (group valueVariable: 'deadline'))  Once: [
26.            ((group valueVariable: 'numYes') >= (group valueVariable: 'numNot'))
27.                 ifTrue: [ voters sendFinalResult: 'Yes' ]
28.                 ifFalse: [ voters sendFinalResult: 'Not' ]]
```

Figure 9 Electronic Voting

Rule 1 (line 9): A voter initializes a vote by invoking his method startVote:deadline: and fixes a deadline. This triggers a multicasted message voteOn:initiator: to all members of the group by passing the initiator of the vote.

Rule 2 (line 19): When the vote initiator receives the votes of the group members he counts them. Positive and negative votes are counted using the global coordination variables NumYes and NumNot. The participant who voted is marked as "has voted" (the variable whoVoted associated with each voter is set to true).

Rule 3 (line 25): Results coming from voters marked as "has voted", or coming after deadline expiration are ignored by the initiator of the vote.

Rule 4 (line 26): Once the deadline has expired the system calculates and sends the result of the vote to all the participants.

As votes that arrived too late and votes sent twice by the same voter should not be counted they are ignored as shown in the rule 3 using the Ignore operator. The Proaction Once rule (rule 4) is used to send the final result of the vote once the deadline of the vote expires. Note that the presented solution does not support the confidentiality of the vote as in [Mins97a]. This situation is not linked to the CoLaS model but is just due to the way this example is expressed. To support vote confidentiality, the result of the vote should be returned to a new object dedicated to this task instead of the initiator.

7 Evaluation of the CoLaS Model

We now evaluate how CoLaS answers the problems and requirements we identified in Section 2. CoLaS evaluates positively for the following properties:

- *Separation of Concerns:* The coordination is not hard-wired any more into the coordinated objects. The coordination and computational aspects are specified separately in distinct entities: groups and participants.
- *Enforcing Encapsulation:* The coordination does not refer to the internals of coordinated objects. It is only expressed using the interface of the coordinated objects. Moreover, a group encapsulates the coordination information (state, behaviour) in a single and identifiable entity.
- *Multi Object Coordination Abstraction:* The coordination is not limited to two objects but to a group of objects. Moreover, the coordination specifies abstractly the different behaviour of the participants in terms of roles and their respective interfaces. Roles allows one to specify the coordination behaviour independently of the effective participant number. Thus roles specify *intentionally* the behaviour of a set of objects.
- *High-Level Abstraction:* The programmer no longer focuses on how to do the coordination but on how to express it. All the low-level operations concerning the coordination are managed by CoLaS. For example programmers do not lock or unlock participants to guarantee their consistency. The group protects participant states from third-party accesses by enforcing automatic locking policies.
- *Dynamic Evolution of Coordination:* The coordination behaviour specification is not rigid any more. CoLaS supports dynamic coordination changes in three distinct aspects: (1) coordination behaviour can be changed dynamically by adding/removing new rules, (2) new coordination groups can be created and destroyed at any time, and (3) new participants can join or leave the group at any time without disturbing other participants.
- *Coordination Specification:* Coordination is expressed declaratively in CoLaS by using rules. Coordination rules are specified by users that incrementally included them into groups. Rules are specified independently of the coordinated entities.

In the current state of CoLaS the following aspects are not supported:

- *Composability of Coordination:* Existing coordination patterns cannot be combined into new ones.
- *Formal Properties:* We do not have yet a formal model of CoLaS that we can use to formally validate properties of the coordination layer.
- *Incremental Definition of Groups*.

CoLaS a Language Independent Model. CoLaS is independent of any language. It is only based on the control of message passing. That's why it can be implemented in languages that provides such a functionality like CLOS, MetaAxa (a reflective extension of Java) or OpenC++ or using code instrumentation for trapping the messages.

8 Related Work and Conclusions

CoLaS. In this paper we introduced the CoLaS coordination model and its implementation in Smalltalk that supports coordination of active objects in concurrent object-oriented systems. CoLaS offers a high-level construct called Coordination Group that specifies and *enforces* the coordination of collaborating active objects by means of rules. Our approach mainly differs from similar approaches in that: (1) the cooperation is specified and enforced within the group; (2) the coordination state includes per-role information necessary to realize coordination; (3) the coordination includes the enforcement of permanent actions depending on the coordination state; (4) the state of participants is taken in account in the coordination; and (5) we support dynamic evolution of the coordination aspect: objects can join and leave the group at any time, coordination rules can be added and removed on the fly, and new groups can be created at run-time.

Related Work. Traditionally the coordination layer of concurrent object-oriented systems is developed using concurrent object-oriented languages. These languages provide only limited support for the specification of an abstraction of the coordination. In the last few years, a new set of so-called *coordination languages*: Linda [Gele92a], Gamma [Bana95a], Manifold [Arba96b], ActorSpace [Agha93c], Objective Linda [Kiel96a] to cite a few, have been developed to support the construction of coordination layers in software system. Due to space limitation we limit ourselves the approaches that support object-oriented concurrent programming and we focus on different comparisons than the ones we already made in Section 2.

In ObjectiveLinda [Kiel96a] objects are *aware* of the existence of a virtual shared space to which they must communicate. The coordination is not *transparent* to the coordinated objects. The same situation occurs with ACT [Aksit94a] contrary to CoLaS and [Frol93a, Mukh95a, Mins97a, Duca98c]. In CoLaS as in most of the approaches [Frol93a, Aksit94a, Mukh95a, Andr96b, Mins97a, Duca98c] the coordination is specified and encapsulated independently of the representation of the entities they coordinate. Concerning other coordination properties, CoLaS is a hybrid model (as in [Frol93a, Aksi92a, Mukh95a, Mins97a]), the coordination is achieved through the cooperation of both objects and a coordinator agent.

Groups behaviour not only constrains the treatment of messages as in [Frol93a, Aksi92a] but also enforces coordinated actions on participants. Such enforcements are done by reacting to certain messages or by *initiating* actions depending on the state of the coordination. In Moses [Mins97a] coordination actions can be enforced too but the actions only affect the receiver of the message (i.e. forward the message once it is received and modify the control state of the receiver). CoLaS coordination actions can be applied to any group participant. Moreover, CoLaS coordination rules may refer to different coordination information including participant states via state predicates. In Moses, coordination policies refer only to the local control state of object who has received the message, and in [Frol93a] they refer only to the state of the Synchronizer.

One the most important aspect of CoLaS is its support for dynamic evolution of the coordination. A group is a complete *dynamic* entity that can be created and destroyed at any time, and in which coordination rules can be added and removed, and participants can join or leave the group at any time. Approaches like [Frol93a, Aksi92a, Mins97a] do not manage the full dynamicity of the coordination.

Acknowledgment. The authors would like to thanks Oscar Nierstrasz, Tamar Richner, Serge Demeyer and the anonymous reviewers for their valuable comments.

References

[Agha93c] G. Agha and C. J. Callsen, ActorSpace: An Open Distributed Programming Paradigm, *Proc.4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, vol. 28, no. 7, 1993, pp. 23-323.

[Aksi92a] M. Aksit and L. Bergmans, Obstacles in Object-Oriented Software Development, *OOPSLA '92, ACM SIGPLAN Notices*, vol. 27, no. 10, Oct. 1992, pp. 341-358.

[Aksi94a] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, Abstracting Object Interactions Using Composition Filters, LNCS 791, 1994, pp. 152-184.

[Alle94c] R. Allen and D. Garlan, Formalizing Architectural Connection, *ICSE'94*, May 1994.

[Andr96b] 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.

[Arba96b] F. Arbab, The IWIM Model for Coordination of Concurrent Activities, *COORDINATION'96*, LNCS 1061, Springer-Verlag, 1996, pp. 34-55.

[Bana95a] J.-P. Banâtre and Daniel Le Métayer, Gamma and the Chemical Reaction Model, *Coordination'95 Workshop*, IC Press, Londres, 1995.

[Bloo79a] T. Bloom, Evaluating Synchronization Mechanisms, *Proceedings of the Seventh Symposium on Operating Systems Principles*, 1979, pp. 24-32.

[Brio96a] J.-P. Briot, An Experiment in Classification and Specialization of Synchronization Schemes, *LNCS*, vol. 1049, Springer-Verlag, 1996, pp. 227-249.

[Duca98c] S. Ducasse and M. Guenter, Coordination of Active Objects by Means of Explicit Connectors, *DEXA workshops*, IEEE Computer Society Press, pp. 572-577.

[Frol93a] S. Frolund and G. Agha, A Language Framework for Multi-Object Coordination, *ECOOP'93*, LNCS 707, Springer-Verlag, July 1993, pp. 346-360.

[Gele92a] D. Gelernter and N. Carriero, Coordination Languages and their significance, *CACM*, vol. 35, no. 2, February 1992.

[Helm85a] D. Helmbold and D. Luckman. Debugging Ada Tasking Programs, *IEEE Software* vol. 2, no 2, March 1985, pp. 47-57.

[Kiel96a] T. Kielmann, Designing a Coordination Model for Open Systems,*COORDINATION'96*, LNCS 1061, Springer-Verlag, 1996, pp. 267-284.

[Lea96a] D. Lea, *Concurrent Programming in Java — Design principles and Patterns*, Addison-Wesley, 1996.

[Lope97a] C.V.Lopez and G. Kiczales, "D: A Language Framework for Distributed Programming", Tech. Rep. TR SPL97-010P9710047, Xerox Parc., 1997.

[Mins97a] N. Minsky and V. Ungureanu, Regulated Coordination in Open Distributed Systems,*COORDINATION'97*, LNCS 1282, Springer-Verlag, 1997, pp. 81-97.

[Mukh95a] M. Mukherji and D. Kafura, *Specification of Multi-Object Coordination Schemes Using Coordinating Environments R Draft*, Virgina Tech, 1995.

[Papa96a] M. Papathomas, ATOM: An Active object model for enhancing reuse in the development of concurrent software, RR 963-I-LSR-2, IMAG-LSR, Grenoble-France, 1996.