

OpenCoLaS a Coordination Framework for CoLaS Dialects

Juan Carlos Cruz

Software Composition Group, University of Berne – Switzerland
cruz@iam.unibe.ch

Abstract. An important family of existing coordination models and languages is based on the idea of trapping the messages exchanged by the coordinated entities and by the specification of rules governing the coordination. No model, including our CoLaS coordination model, justifies clearly the reason of their coordination rules. Why these rules and not others? Are they all necessary? These are questions that remain still open. In order to try to provide an answer, in particular for the CoLaS model, we propose in this paper OpenCoLaS, a framework for building CoLaS coordination dialects. The OpenCoLaS framework allows to experiment with the definition of coordination rules.

1. Introduction

CoLaS is a coordination model and language for distributed active objects based on the notion of Coordination Groups. A Coordination Group is an entity that specifies, controls and enforces coordination between groups of distributed active objects. A Coordination Group specifies the Roles that distributed active objects may play in the group and the set of Coordination Rules that govern the coordination of the group. Since our first presentation of CoLaS in [1] the coordination model has changed significantly. These changes concern mainly the Coordination Rules: new rules have been introduced, some others have disappeared. These changes were motivated by the goal of obtaining a clearer separation of coordination and computation in the model. Questions such as “Why these rules and not others?”, “Are all these rules necessary?” appear all the time and remain still open. In order to try to provide answers, we propose in this paper to define OpenCoLaS, a framework to experiment with the specification of the CoLaS model. Our idea is to “open” the CoLaS model and language in a way that allows to modify and to introduce new Coordination Rules. By opening the CoLaS model we will be able to experiment with its specification.

This paper is organized in the following way: Section 2 introduces shortly the current version of the CoLaS coordination model and language; section 3 shows an example of the use of CoLaS: The Electronic Vote [4]; section 4 introduces the OpenCoLaS framework; and finally section 5 sets out some reflections on the CoLaS model related to the open questions mentioned before and presents some related work.

2. The CoLaS Coordination Model and Language

The CoLaS coordination model and language [1][2] is composed of two kinds of entities: the Coordination Groups and the Coordination Group Participants (Participants in the following).

2.1 The Participants

Participants in CoLaS are atomic distributed active objects (distributed objects in the following). They are active objects because they are concurrent objects that have control over concurrent method invocations. They are distributed because physically they may run on different processors or machines. And they are atomic because they process invocations atomically.

2.2 Coordination Groups

Coordination Groups (CGs in the following) are entities that specify, control and enforce the coordination of a group of Participants. The primary tasks of CGs 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) in Participants based on the coordination state.

Coordination Specification

A CG is composed of three elements: the Roles Specification, the Coordination State, and the Coordination Rules.

- *The Roles Specification*: defines the Roles that Participants may play in the group. A Role identifies abstractly a set of Participants sharing the same coordination behavior (i.e. same Coordination Rules).
- *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 in the system (i.e. historical information), etc. The Coordination State is specified by defining variables.
- *The Coordination Rules*: define the rules governing the coordination of the CG. We define 4 kinds of Coordination Rules in CoLas:

Cooperation Rules define cooperation actions between participants. They specify actions that Participants should do when they receive specific method invocations. Participants react to these method invocations only during the time they play the Role. Cooperation Rules have the form: <Role> defineBehavior: <Message> as: [<Coordination Actions>]. The <Coordination Actions> include actions that manipulate the coordination state and method invocations to Participants.

Synchronization Rules define synchronization constraints on method invocations received by Participants. The Synchronization Rules have the form:

<Role> ignore: <Message> if: [<Synchronization Conditions>] and <Role> disable: <Message> do: [<Coordination Actions>]. In the ignore rule the method invocation <Message> received is ignored if the <Synchronization Conditions> hold. The <Synchronization Conditions> are conditions referring to the Coordination State or conditions referring to the arguments of the method invocations. In the disable rule the method invocation received is disabled (processed later) and the <Coordination Actions> executed. The <Coordination Actions> are the same that in the Cooperation Rules.

Interception Rules define actions that modify the Coordination State at different moments during the handling of the method invocations by the Participants. Interception Rules have the form <Role> <Message> <Interception Point> do: <State Actions>. There are three different kinds of interceptions (InterceptAtArrival, InterceptBeforeExecution, and InterceptAfterExecution) according to the moment at which the method invocation <Message> should be intercepted. The <State Actions> are actions that modify the Coordination State.

Proactive Rules define actions that should be executed by the CG based on the coordination state, and independently of the method invocations received by the Participants. Proactive Rules have the form Proaction conditions: <Proaction Conditions> do: <Coordination Actions>. The <Proaction Conditions> are conditions referring the Coordination State. The <Coordination Actions> are the same as in the Cooperation Rules. The evaluation of the Proactive Rules is done non-deterministically by the CG.

3. An Example – The Electronic Vote

To introduce the current version of the CoLaS model we will use the same example we used in the presentation of the original CoLaS coordination model [1]: The Electronic Vote (originally introduced in [4]). We have selected this example because it will help us to compare Coordination Rules between the original model and the current CoLaS model.

Coordination Specification

The Electronic Vote Coordination Group defines a unique role Voters (Fig. 1, line 1). Every participant of the role Voters should at least be able to respond to the opinion: invocation request (Fig. 1, line 2). Eight Coordination Rules govern the coordination of the group: rules 1, 2, 3 and 4 specify cooperation rules; rules 5 and 6 interception rules; and rules 7 and 8 synchronization rules.

Rule 1 (line 8) Rule 2 (line 12) Rule 3 (line 15) Rule 4 (line 21) :define The Electronic Vote protocol. The vote process starts with a startVote: message is sent by a Participant, the startVote: behavior (rule 1) defines that a message voteOn: initiator: is sent to all the Participants of the Voters role. The voteOn: initiator: behavior (rule 2) defines that each voter according to his or her opinion sends a message vote: to the initiator of the vote process. The vote:

behavior (rule 3) defines the counting of the received votes. Finally the `stopVote`: behavior (rule 4) defines that the result of the vote process is sent to all the voters when the initiator decides to stop the period of vote.

```
1 adminVote defineRoles: #(Voters).
2 Voters defineInterface: #(opinion:).
3
4 adminVote defineVariables: #(voteInProgress=false, votePeriodExpired=false).
5 adminVote defineVariables: #(numYes=0, numNot=0).
6 Voters defineVariables: #(hasVoted=false).
7.
8 [1] Voters defineBehavior: 'startVote: anIssue' as:
9   [group voteInProgress: true.
10    Voters voteOn: anIssue initiator: receiver ].
11
12 [2] Voters defineBehavior: 'voteOn: anIssue initiator: aVoter' as:
13   [aVoter vote: (self opinion: anIssue) ].
14
15 [3] Voters defineBehavior: 'vote: aVote' as:
16   [aVote = 'Yes'
17    ifTrue: [ group incrVariable: #numYes ]
18    ifFalse: [ group incrVariable: #numNot ].
19    sender hasVoted: true.]
20
21 [4] Voters defineBehavior: 'stopVote' as:
22   [group numYes >= group NumNot
23    ifTrue: [ Voters voteResult: 'Yes' ]
24    ifFalse: [ Voters voteResult: 'Not' ].
25
26 [5] Voters interceptBeforeExecution: 'stopVote' do:
27   [group votePeriodExpired: true ].
28
29 [6] Voters interceptAfterExecution: 'stopVote' do:
30   [group voteInProgress: false.
31    group votePeriodExpired: false ].
32
33 [7] Voters ignore: 'vote: aVote' if:
34   [group votePeriodExpired || sender hasVoted ]
35
36 [8] Voters disable: 'startVote: anIssue' do:
37   [group voteInProgress ]
```

Fig. 1. The Electronic Vote in CoLaS.

Rule 5 (line 26) and Rule 6 (line 29) define Interception Rules. They specify actions on the Coordination State. Rule 5 specifies that the vote period is closed before the execution of the `stopVote`: message. Rule 6 defines that a new vote process may start after the execution of the `stopVote`:message.

Rule 7 (line 33) defines that votes received after the end of the period of vote or votes received from Participants that have already voted are ignored.

Rule 8 (line 36) defines that new requests for starting a vote process are disabled (processed later) if there is actually one vote process occurring in the system.

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 (lines 9, 17, 18, 22, 27, 30, 31, 34 and 37). The `sender` pseudo-variable refers to the Participant that sent the message (line 34), and the `receiver` pseudo-variable refers to the Participant handling the message (line 10).

4 The OpenCoLaS Framework

OpenCoLaS is a framework that allows to specify Coordination Rules for CoLaS like coordination models. The OpenCoLaS framework defines an abstract class named `CoordinationRule` as the root of all possible Coordination Rules. There are two possible types of Coordination Rules (abstract subclasses of `CoordinationRule`): Reactive Coordination Rules (`ReactiveCoordinationRule`) and Proactive Coordination Rules (`ProactiveCoordinationRule`).

4.1 Reactive Coordination Rules

Reactive Coordination Rules specify actions that should be triggered on method invocation requests received by the Participants. The `ReactiveCoordinationRule` class offers an instance creation method `defineRule: <RuleName> semantic: <Rule Semantic Actions>` used to create concrete Reactive Coordination rules. The `<Rule Semantic Actions>` define actions over a received method invocation request: it is possible to define a new method invocation with new argument values, or to transform the method invocation in a special `NoMessage` method invocation that indicates that the received method invocation should not continue to be processed. The `<Rule Semantic Actions>` actions also specify actions on the participant's messages mailbox (a mailbox is the place where method invocations are stored in Participants when they can not process them): like to remove a method invocation from the mailbox or to include a method invocation into the mailbox. To refer to the received method invocation and to the messages mailbox in the `<Rule Semantic Actions>`, there are two simple accessor methods: `message` and `mailbox`. Finally, the last action of the `<Rule Semantic Actions>` actions should always be to return a method invocation: the same received, a new, or a `NoMessage` method invocation.

The `ReactiveCoordinationRule` offers a second creation method `defineRule: <RuleName>` without a semantics, in this case the `semantic` by default corresponds to a return the same received method invocation. To illustrate the specification of Reactive Coordination Rules in the OpenCoLaS framework we use the CoLaS coordination model. In the CoLaS coordination model we have 5 types of rules that depend for their application on the method invocations received by the Participants. They are: `Ignore`, `Disable`, `InterceptAtArrival`, `InterceptBeforeExecution`, and `InterceptAfterExecution`. Fig. 2 illustrates the way in which the CoLaS Reactive Coordination Rules are specified in the OpenCoLaS framework.

```

1 [1]ReactiveCoordinationRule
2   defineRule: #Ignore
3   semantic: [^NoMessage new ].
4
5 [2]ReactiveCoordinationRule
6   defineRule: #Disable
7   semantic: [self mailbox put: self message.
8             ^NoMessage new ].
9
10 [3]ReactiveCoordinationRule
11   defineRule:#InterceptAtArrival
12
13 [4]ReactiveCoordinationRule
14   defineRule: #InterceptBeforeExecution
15
17 [5]ReactiveCoordinationRule
18   defineRule: #InterceptAfterExecution

```

Fig. 2. CoLaS Reactive Coordination Rules in OpenCoLaS.

Ignore (line 1): the Ignore rule specifies the return of a `NoMessage` message. The method invocation should not continue to be processed.

Disable (line 5): the Disable rule specifies that the method invocation received should be put into the participant's messages mailbox and thus processed later. The return of a `NoMessage` indicates that the method invocation should not continue to be processed.

InterceptAtArrival (line 10), InterceptBeforeExecution (line 13) and InterceptAfterExecution (line 17): specify that no action should be done on the received invocation request message.

```

1 [5] InterceptBeforeExecution
2   message: 'stopVote'
3   actions: [group votePeriodExpired: true ]
4   entryPoint: OpenCoLaS AtAccept ]
5
6 [6] InterceptAfterExecution
7   message: 'stopVote'
8   actions: [group voteInProgress: false.
9             group votePeriodExpired: false]
10  entryPoint: OpenCoLaS AtComplete ]
11
12 [7] Ignore
13  message: 'vote: aVote'
14  conditions: [ group votePeriodExpired || sender hasVoted ]
15  entryPoint: OpenCoLaS AtReceive
16
17 [8] Disable
18  message: 'startVote: anIssue'
19  actions: [ group voteInProgress ]
20  entryPoint: OpenCoLaS AtAccept.

```

Fig. 3. Creation of CoLaS Reactive Coordination Rules in OpenCoLaS

In order to complete the specification of the CoLaS reactive rules it is necessary to specify during the creation of the rule instances: the <Reaction Message> to which the rule is supposed to react (**message:**), the <Coordination Conditions> that determine the applicability of the rule (**conditions:**), the <Coordination Actions> that should be executed when the rule is applied (**actions:**), and the <Entry Point> at which the rule should be verified (**entryPoint:**). The OpenCoLaS framework specifies four different entry points: **AtReceive**, **AtAccept**, **AtSend** and **AtComplete**.

The OpenCoLaS framework specifies default values for the <Coordination Conditions> and the <Coordination Actions>. For the <Coordination Conditions> a block [true] that always validates, and for <Coordination Actions> an empty block [] with no actions to be executed. Fig.3 illustrates the way in which CoLaS rule instances are created in OpenCoLaS for the Electronic Vote example.

The different <Entry Point> (Fig 4.) correspond to 4 different moments during the internal processing of method invocations requests by the Participants: **AtReceive** (when the method invocation arrives to the object), **AtAccept** (when the method invocation is selected from the messages mailbox and just before it is executed), **AtSend** (when a method invocation is sent to another object and actually before it has been received by the other object), and **AtComplete** (after the method invocation has been executed).

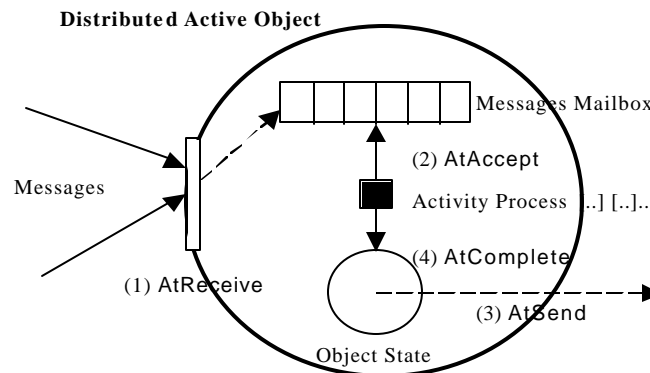


Fig. 4. OpenCoLaS <Entry Point>s.

4.2 Proactive Coordination Rules

Proactive Coordination Rules specify actions that should be triggered based on the Coordination State. The **ProactiveCoordinationRule** class in the OpenCoLaS framework offers an instance creation method **defineRule: <RuleName>** to create subclasses of Proactive Coordination rules. During the creation of Proactive Coordination Rules instances it is necessary to specify: the <Coordination Conditions> that determine the applicability of the rule (**conditions:**), and the <Coordination Actions> the actions that should be executed when the rule is applied (**actions:**). In CoLaS the evaluation and enforcement of the Proactive Coordination

Rules is done by the CG. Fig. 5 illustrates the way in which the CoLaS Proactive Coordination Rules are specified in the OpenCoLaS framework.

```
1 [1]ProactiveCoordinationRule
2   defineRule: #Proaction
3
4 [2] Proaction
5   conditions: [group votePeriodExpired ]
6   actions: [group numYes >= group NumNot
7             ifTrue: [ Voters voteResult: 'Yes' ]
8             ifFalse: [ Voters voteResult: 'Not' ]. ]
9   group votePeriodExpired: false ]
```

Fig. 5. CoLaS Proactive Coordination Rules in OpenCoLaS.

Rule 2 (line 4) defines that when the vote period expires the result of the vote should be sent to all the Voters.

5 Conclusions and Related Work

One of the main results of this work is the conclusion that the three Interception Rules specified in CoLaS can be reduced to a unique Interception Rule. We can see in Fig. 2 that the three Interception rules specified have the same <Rule Semantic Actions> and the same <Coordination Actions>. The only difference between the three rules is that they are evaluated at different entry points. We can conclude that a unique `InterceptAt` rule is necessary in the CoLaS model. It is possible to include a new Interception Rule to intercept messages at the `AtSend` entry point, actually in CoLaS there is not Interception Rule associated with this entry point.

Concerning Proactive Coordination Rules we arrived at the conclusion that the two Proactive rules: `Always` and `Once` in the original CoLaS model were not cleanly defined. The semantics of the two rules were not clear, they could have been reduced to one as we did in the current version of CoLaS.

The most important related work are Synchronizers [3] and Moses [4]. For a complete CoLaS related work refer to [1][2]. Concerning OpenCoLaS there is not much work on meta coordination models (particularly in object oriented coordination models). From our point of view meta coordination models should constitute a field of research on its own in the coordination area.

References

1. J.C.Cruz and S. Ducasse, *A Group Based Approach for Coordinating Active Objects*, Coordination'99, LNCS 1594, Springer Verlag, pp. 355-370.
2. 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.
3. S. Frolund, *Coordinating Distributed Objects*, MIT Press, 1996.
4. N.Minsky and V. Ungureanu, *Regulated Coordination in Open Distributed Systems*, COORDINATION'97, LNCS 1282, Springer-Verlag, 1997, pp. 81-97.